Introduction to computing with finite difference methods

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Aug 19, 2014

Contents

Finite difference methods for partial differential equations (PDEs) e
range of concepts and tools that can be introduced and illustrated in the
of simple ordinary differential equation (ODE) examples. This is what v

ıe present document. By first working with ODEs, we keep the mathematical roblems to be solved as simple as possible (but no simpler), thereby allowing ıll focus on understanding the key concepts and tools. The choice of topics ı the forthcoming treatment of ODEs is therefore solely dominated by what ırries over to numerical methods for PDEs.

Theory and practice are primarily illustrated by solving the very simple)DE $u' = -au$, $u(0) = I$, where $a > 0$ is a constant, but we also address the eneralized problem $u' = -a(t)u + b(t)$ and the nonlinear problem $u' = f(u, t)$. he following topics are introduced:

- How to think when constructing finite difference methods, with special focus on the Forward Euler, Backward Euler, and Crank-Nicolson (midpoint) schemes
- How to formulate a computational algorithm and translate it into Python code
- How to make curve plots of the solutions
- How to compute numerical errors
- How to compute convergence rates
- How to verify an implementation and automate verification through nose tests in Python
- How to structure code in terms of functions, classes, and modules
- How to work with Python concepts such as arrays, lists, dictionaries, lambda functions, functions in functions (closures), doctests, unit tests, command-line interfaces, graphical user interfaces
- How to perform array computing and understand the difference from scalar computing
- How to conduct and automate large-scale numerical experiments
- How to generate scientific reports
- How to uncover numerical artifacts in the computed solution
- How to analyze the numerical schemes mathematically to understand why artifacts occur
- How to derive mathematical expressions for various measures of the error in numerical methods, frequently by using the `sympy` software for symbolic computation
- Introduce concepts such as finite difference operators, mesh (grid), mesh functions, stability, truncation error, consistency, and convergence

- Present additional methods for the general nonlinear ODE $u' =$ which is either a scalar ODE or a system of ODEs
- How to access professional packages for solving ODEs
- How the model equation $u' = -au$ arises in a wide range of pheno physics, biology, and finance

The exposition in a nutshell.

Everything we cover is put into a practical, hands-on context. All mathe ics is translated into working computing codes, and all the mathema theory of finite difference methods presented here is motivated fr strong need to understand strange behavior of programs. Two fundam questions saturate the text:

- How to we solve a differential equation problem and produce num
- How to we trust the answer?

1 Finite difference methods

Goal.

We explain the basic ideas of finite difference methods using a si ordinary differential equation $u' = -au$ as primary example. Emp is put on the reasoning when discretizing the problem and introdu of key concepts such as mesh, mesh function, finite difference appro tions, averaging in a mesh, derivation of algorithms, and discrete ope notation.

1.1 A basic model for exponential decay

Our model problem is perhaps the simplest ordinary differential equation

$$u'(t) = -au(t),$$

Here, $a > 0$ is a constant and $u'(t)$ means differentiation with respect t This type of equation arises in a number of widely different phenomen some quantity u undergoes exponential reduction. Examples include rad decay, population decay, investment decay, cooling of an object, pressu in the atmosphere, and retarded motion in fluids (for some of these

can be negative as well), see Section 8 for details and motivation. We have hosen this particular ODE not only because its applications are relevant, but ven more because studying numerical solution methods for this simple ODE ives important insight that can be reused in much more complicated settings, particular when solving diffusion-type partial differential equations.

The analytical solution of the ODE is found by the method of separation of ariables, which results in

$$u(t) = Ce^{-at},$$

or any arbitrary constant C. To formulate a mathematical problem for which here is a unique solution, we need a condition to fix the value of C. This ondition is known as the *initial condition* and stated as $u(0) = I$. That is, we now the value I of u when the process starts at $t = 0$. The exact solution is hen $u(t) = Ie^{-at}$.

We seek the solution $u(t)$ of the ODE for $t \in (0, T]$. The point $t = 0$ is not ncluded since we know u here and assume that the equation governs u for $t > 0$. he complete ODE problem then reads: find $u(t)$ such that

$$u' = -au, \ t \in (0, T], \quad u(0) = I\,. \tag{1}$$

his is known as a *continuous problem* because the parameter t varies continuously om 0 to T. For each t we have a corresponding $u(t)$. There are hence infinitely any values of t and $u(t)$. The purpose of a numerical method is to formulate a orresponding *discrete* problem whose solution is characterized by a finite number f values, which can be computed in a finite number of steps on a computer.

.2 The Forward Euler scheme

olving an ODE like (1) by a finite difference method consists of the following our steps:

1. discretizing the domain,
2. fulfilling the equation at discrete time points,
3. replacing derivatives by finite differences,
4. formulating a recursive algorithm.

tep 1: Discretizing the domain. The time domain $[0, T]$ is represented y a finite number of $N_t + 1$ points

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,. \tag{2}$$

he collection of points $t_0, t_1, \ldots, t_{N_t}$ constitutes a *mesh* or *grid*. Often the esh points will be uniformly spaced in the domain $[0, T]$, which means that

the spacing $t_{n+1} - t_n$ is the same for all n. This spacing is often denote in this case $t_n = n\Delta t$.

We seek the solution u at the mesh points: $u(t_n)$, $n = 1, 2, \ldots, N_t$. N u^0 is already known as I. A notational short-form for $u(t_n)$, which will extensively, is u^n. More precisely, we let u^n be the *numerical approximatio* exact solution $u(t_n)$ at $t = t_n$. The numerical approximation is a *mesh f* here defined only at the mesh points. When we need to clearly dist between the numerical and the exact solution, we often place a sub on the exact solution, as in $u_e(t_n)$. Figure 1 shows the t_n and u_n po $n = 0, 1, \ldots, N_t = 7$ as well as $u_e(t)$ as the dashed line. The goal of a nu method for ODEs is to compute the mesh function by solving a finit *algebraic equations* derived from the original ODE problem.

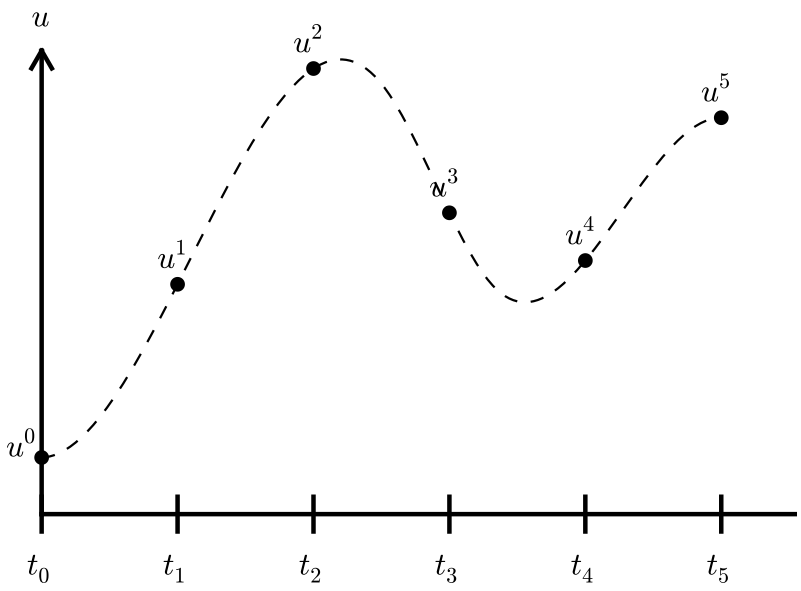
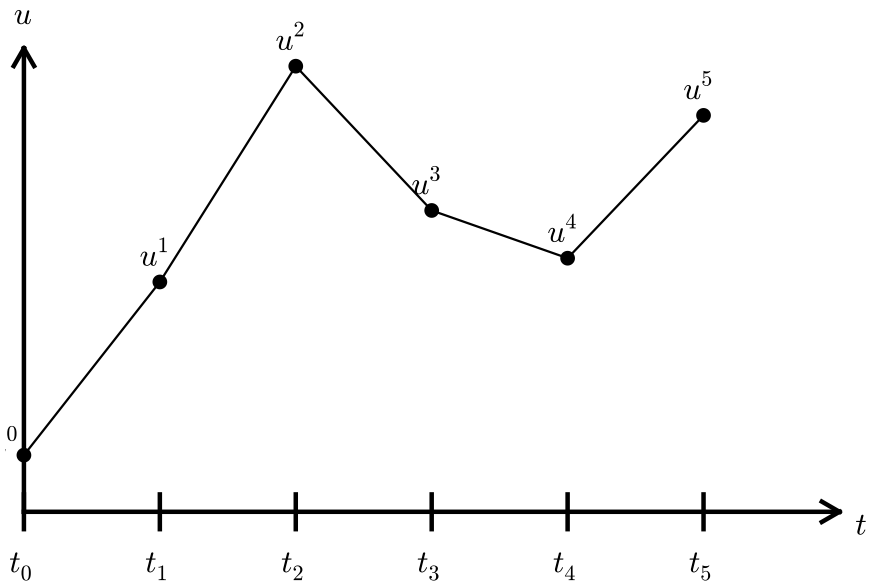


Figure 1: Time mesh with discrete solution values.

Since finite difference methods produce solutions at the mesh points is an open question what the solution is between the mesh points. One methods for interpolation to compute the value of u between mesh poir simplest (and most widely used) interpolation method is to assume that linearly between the mesh points, see Figure 2. Given u^n and u^{n+1}, the u at some $t \in [t_n, t_{n+1}]$ is by linear interpolation

$$u(t) \approx u^n + \frac{u^{n+1} - u^n}{t_{n+1} - t_n}(t - t_n)\,.$$



igure 2: Linear interpolation between the discrete solution values (dashed ırve is exact solution).

tep 2: Fulfilling the equation at discrete time points. The ODE is ıpposed to hold for all $t \in (0, T]$, i.e., at an infinite number of points. Now we ɛlax that requirement and require that the ODE is fulfilled at a finite set of iscrete points in time. The mesh points $t_0, t_1, \ldots, t_{N_t}$ are a natural (but not ıe only) choice of points. The original ODE is then reduced to the following N_t equations:

$$u'(t_n) = -au(t_n), \quad n = 0, \ldots, N_t\,. \tag{4}$$

tep 3: Replacing derivatives by finite differences. The next and most ssential step of the method is to replace the derivative u' by a finite differ- ıce approximation. Let us first try a one-sided difference approximation (see igure 3),

$$u'(t_n) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,. \tag{5}$$

ıserting this approximation in (4) results in

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -au^n, \quad n = 0, 1, \ldots, N_t - 1\,. \tag{6}$$

Later it will be absolutely clear that if we want to compute the solu to time level N_t, we only need (4) to hold for $n = 0, \ldots, N_t - 1$ since $n = N_t - 1$ creates an equation for the final value u^{N_t}.

Equation (6) is the discrete counterpart to the original ODE prob and often referred to as *finite difference scheme* or more generally as the *equations* of the problem. The fundamental feature of these equations they are *algebraic* and can hence be straightforwardly solved to prod mesh function, i.e., the values of u at the mesh points $(u^n, n = 1, 2, \ldots$

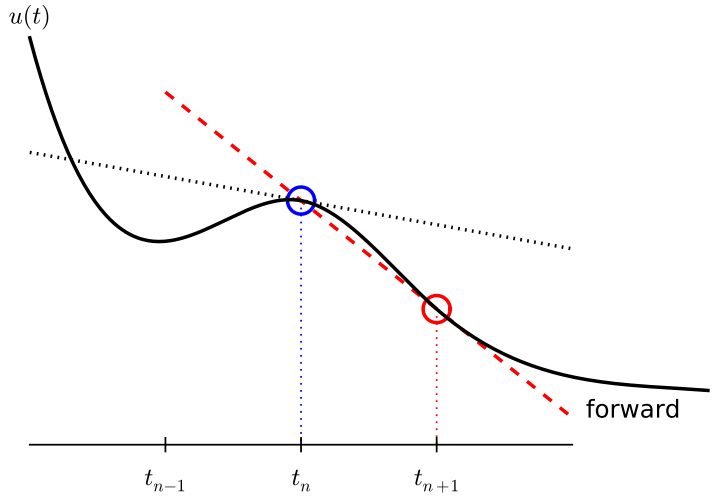


Figure 3: Illustration of a forward difference.

Step 4: Formulating a recursive algorithm. The final step is to the computational algorithm to be implemented in a program. The ke vation here is to realize that (6) can be used to compute u^{n+1} if u^n is Starting with $n = 0$, u^0 is known since $u^0 = u(0) = I$, and (6) gives an ɛ for u^1. Knowing u^1, u^2 can be found from (6). In general, u^n in (6) assumed known, and then we can easily solve for the unknown u^{n+1}:

$$u^{n+1} = u^n - a(t_{n+1} - t_n)u^n\,.$$

We shall refer to (7) as the Forward Euler (FE) scheme for our model ı From a mathematical point of view, equations of the form (7) are kı *difference equations* since they express how differences in u, like $u^{n+1} - u^n$ with n. The finite difference method can be viewed as a method for tu differential equation into a difference equation.

Computation with (7) is straightforward:

$$
\begin{aligned}
u_0 &= I,\\
u_1 &= u^0 - a(t_1 - t_0)u^0 = I(1 - a(t_1 - t_0)),\\
u_2 &= u^1 - a(t_2 - t_1)u^1 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1)),\\
u^3 &= u^2 - a(t_3 - t_2)u^2 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1))(1 - a(t_3 - t_2)),
\end{aligned}
$$

nd so on until we reach u^{N_t}. Very often, $t_{n+1} - t_n$ is constant for all n, so e can introduce the common symbol Δt for the time step: $\Delta t = t_{n+1} - t_n$, $= 0, 1, \ldots, N_t - 1$. Using a constant time step Δt in the above calculations ives

$$
\begin{aligned}
u_0 &= I,\\
u_1 &= I(1 - a\Delta t),\\
u_2 &= I(1 - a\Delta t)^2,\\
u^3 &= I(1 - a\Delta t)^3,\\
&\vdots\\
u^{N_t} &= I(1 - a\Delta t)^{N_t}\,.
\end{aligned}
$$

his means that we have found a closed formula for u^n, and there is no need o let a computer generate the sequence $u^1, u^2, u^3, \ldots$. However, finding such formula for u^n is possible only for a few very simple problems, so in general nite difference equations must be solved on a computer.

As the next sections will show, the scheme (7) is just one out of many lternative finite difference (and other) methods for the model problem (1).

.3 The Backward Euler scheme

here are several choices of difference approximations in step 3 of the finite ifference method as presented in the previous section. Another alternative is

$$
u'(t_n) \approx \frac{u^n - u^{n-1}}{t_n - t_{n-1}}\,. \tag{8}
$$

ince this difference is based on going backward in time (t_{n-1}) for information, is known as the Backward Euler difference. Figure 4 explains the idea.

Inserting (8) in (4) yields the Backward Euler (BE) scheme:

$$
\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -au^n\,. \tag{9}
$$

Ve assume, as explained under step 4 in Section 1.2, that we have computed $^0, u^1, \ldots, u^{n-1}$ such that (9) can be used to compute u^n. For direct similarity

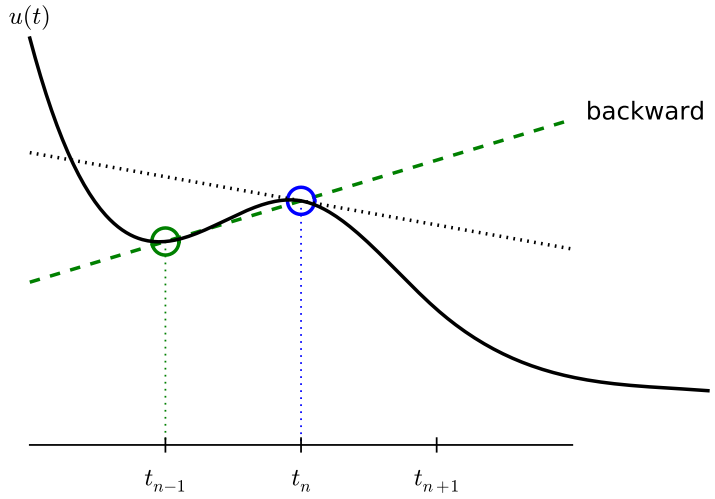


Figure 4: Illustration of a backward difference.

with the Forward Euler scheme (7) we replace n by $n+1$ in (9) and s the unknown value u^{n+1}:

$$
u^{n+1} = \frac{1}{1 + a(t_{n+1} - t_n)}u^n\,.
$$

1.4 The Crank-Nicolson scheme

The finite difference approximations used to derive the schemes (7) and both one-sided differences, known to be less accurate than central (or m differences. We shall now construct a central difference at $t_{n+1/2} = \frac{1}{2}(t_n$ or $t_{n+1/2} = (n+\frac{1}{2})\Delta t$ if the mesh spacing is uniform in time. The approx reads

$$
u'(t_{n+\frac{1}{2}}) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,.
$$

Note that the fraction on the right-hand side is the same as for the I Euler approximation (5) and the Backward Euler approximation (8) replaced by $n+1$). The accuracy of this fraction as an approximatio derivative of u depends on *where* we seek the derivative: in the cente interval $[t_n, t_{n+1}]$ or at the end points.

With the formula (11), where u' is evaluated at $t_{n+1/2}$, it is na demand the ODE to be fulfilled at the time points *between* the mesh p

$$
u'(t_{n+\frac{1}{2}}) = -au(t_{n+\frac{1}{2}}), \quad n = 0, \ldots, N_t - 1\,.
$$

Using (11) in (12) results in

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -au^{n+\frac{1}{2}}, \tag{13}$$

here $u^{n+\frac{1}{2}}$ is a short form for $u(t_{n+\frac{1}{2}})$. The problem is that we aim to compute
^{n} for integer n, implying that $u^{n+\frac{1}{2}}$ is not a quantity computed by our method.
must therefore be expressed by the quantities that we actually produce, i.e.,
ıe numerical solution at the mesh points. One possibility is to approximate
$^{n+\frac{1}{2}}$ as an arithmetic mean of the u values at the neighboring mesh points:

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1})\,. \tag{14}$$

sing (14) in (13) results in

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a\frac{1}{2}(u^n + u^{n+1})\,. \tag{15}$$

igure 5 sketches the geometric interpretation of such a centered difference.

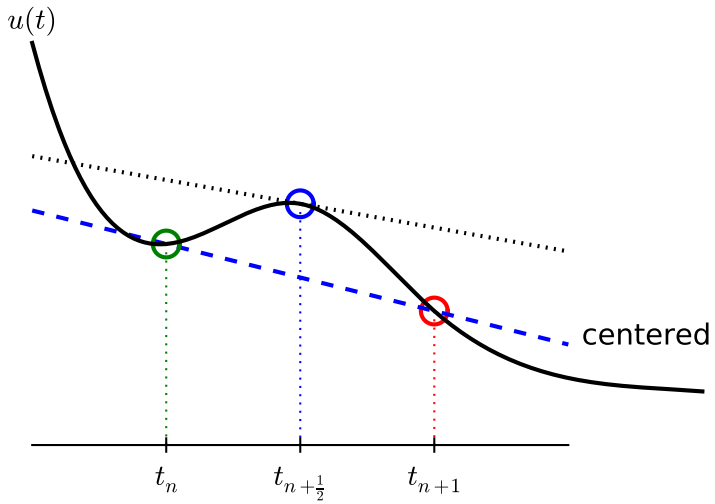


Figure 5: Illustration of a centered difference.

We assume that u^n is already computed so that u^{n+1} is the unknown, which
e can solve for:

$$u^{n+1} = \frac{1-\frac{1}{2}a(t_{n+1}-t_n)}{1+\frac{1}{2}a(t_{n+1}-t_n)}u^n\,. \tag{16}$$

he finite difference scheme (16) is often called the Crank-Nicolson (CN) scheme
r a midpoint or centered scheme.

1.5 The unifying θ-rule

The Forward Euler, Backward Euler, and Crank-Nicolson schemes can b
lated as one scheme with a varying parameter θ:

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a(\theta u^{n+1} + (1-\theta)u^n)\,.$$

Observe:

- $\theta = 0$ gives the Forward Euler scheme
- $\theta = 1$ gives the Backward Euler scheme, and
- $\theta = \frac{1}{2}$ gives the Crank-Nicolson scheme.
- We may alternatively choose any other value of θ in $[0, 1]$.

As before, u^n is considered known and u^{n+1} unknown, so we solve for th

$$u^{n+1} = \frac{1-(1-\theta)a(t_{n+1}-t_n)}{1+\theta a(t_{n+1}-t_n)}\,.$$

This scheme is known as the θ-rule, or alternatively written as the "the

Derivation.

We start with replacing u' by the fraction

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n},$$

in the Forward Euler, Backward Euler, and Crank-Nicolson schemes.
we observe that the difference between the methods concerns which
this fraction approximates the derivative. Or in other words, at w
point we sample the ODE. So far this has been the end points o
midpoint of $[t_n, t_{n+1}]$. However, we may choose any point $\tilde{t} \in [t_n,$
The difficulty is that evaluating the right-hand side $-au$ at an arbi
point faces the same problem as in Section 1.4: the point value mu
expressed by the discrete u quantities that we compute by the scheme
u^n and u^{n+1}. Following the averaging idea from Section 1.4, the val
u at an arbitrary point $\tilde{t}$ can be calculated as a *weighted average*, w
generalizes the arithmetic mean $\frac{1}{2}u^n + \frac{1}{2}u^{n+1}$. If we express $\tilde{t}$ as a weig
average

$$t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_n,$$

where $\theta \in [0, 1]$ is the weighting factor, we can write

$$u(\tilde{t}) = u(\theta t_{n+1} + (1-\theta)t_n) \approx \theta u^{n+1} + (1-\theta)u^n\,.$$

We can now let the ODE hold at the point $\tilde{t} \in [t_n, t_{n+1}]$, approximate u' by the fraction $(u^{n+1} - u^n)/(t_{n+1} - t_n)$, and approximate the right-hand side $-au$ by the weighted average (19). The result is (17).

.6 Constant time step

ll schemes up to now have been formulated for a general non-uniform mesh in me: $t_0, t_1, \ldots, t_{N_t}$. Non-uniform meshes are highly relevant since one can use iany points in regions where u varies rapidly, and save points in regions where is slowly varying. This is the key idea of *adaptive* methods where the spacing f the mesh points are determined as the computations proceed.

However, a uniformly distributed set of mesh points is very common and ufficient for many applications. It therefore makes sense to present the finite ifference schemes for a uniform point distribution $t_n = n\Delta t$, where Δt is the onstant spacing between the mesh points, also referred to as the *time step*. The esulting formulas look simpler and are perhaps more well known.

Summary of schemes for constant time step.

$$u^{n+1} = (1 - a\Delta t)u^n \qquad \text{Forward Euler} \qquad (20)$$

$$u^{n+1} = \frac{1}{1 + a\Delta t}u^n \qquad \text{Backward Euler} \qquad (21)$$

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n \qquad \text{Crank-Nicolson} \qquad (22)$$

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a\Delta t}u^n \qquad \text{The } \theta - \text{rule} \qquad (23)$$

Not surprisingly, we present these three alternative schemes because they ave different pros and cons, both for the simple ODE in question (which can asily be solved as accurately as desired), and for more advanced differential quation problems.

Test the understanding.

At this point it can be good training to apply the explained finite difference discretization techniques to a slightly different equation. Exercise 10 is therefore highly recommended to check that the key concepts are understood.

1.7 Compact operator notation for finite differences

Finite difference formulas can be tedious to write and read, especially for tial equations with many terms and many derivatives. To save space and reader of the scheme to quickly see the nature of the difference approxi we introduce a compact notation. A forward difference approximation is by the D_t^+ operator:

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \approx \frac{d}{dt}u(t_n)\,.$$

The notation consists of an operator that approximates differentiati respect to an independent variable, here t. The operator is built of the sy with the variable as subscript and a superscript denoting the type of di The superscript $^+$ indicates a forward difference. We place square b around the operator and the function it operates on and specify the mes where the operator is acting, by a superscript.

The corresponding operator notation for a centered difference and a b difference reads

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \approx \frac{d}{dt}u(t_n),$$

and

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx \frac{d}{dt}u(t_n)\,.$$

Note that the superscript $^-$ denotes the backward difference, while no sup implies a central difference.

An averaging operator is also convenient to have:

$$[\overline{u}^t]^n = \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) \approx u(t_n)$$

The superscript t indicates that the average is taken along the time coo The common average $(u^n + u^{n+1})/2$ can now be expressed as $[\overline{u}^t]^{n+\frac{1}{2}}$. also spatial coordinates enter the problem, we need the explicit specific the coordinate after the bar.)

The Backward Euler finite difference approximation to $u' = -au$ written as follows utilizing the compact notation:

$$[D_t^- u]^n = -au^n\,.$$

In difference equations we often place the square brackets around th equation, to indicate at which mesh point the equation applies, since ea is supposed to be approximated at the same point:

$$[D_t^- u = -au]^n\,.$$

The Forward Euler scheme takes the form

$$[D_t^+ u = -au]^n, \tag{29}$$

hile the Crank-Nicolson scheme is written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}}\,. \tag{30}$$

Question.

Apply (25) and (27) and write out the expressions to see that (30) is indeed the Crank-Nicolson scheme.

The θ-rule can be specified by

$$[\bar{D}_t u = -a\overline{u}^{t,\theta}]^{n+\theta}, \tag{31}$$

we define a new time difference

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{t^{n+1} - t^n}, \tag{32}$$

nd a *weighted averaging operator*

$$[\overline{u}^{t,\theta}]^{n+\theta} = (1-\theta)u^n + \theta u^{n+1} \approx u(t_{n+\theta}), \tag{33}$$

here $\theta \in [0,1]$. Note that for $\theta = \frac{1}{2}$ we recover the standard centered difference nd the standard arithmetic mean. The idea in (31) is to sample the equation t $t_{n+\theta}$, use a skew difference at that point $[\bar{D}_t u]^{n+\theta}$, and a skew mean value. n alternative notation is

$$[D_t u]^{n+\frac{1}{2}} = \theta[-au]^{n+1} + (1-\theta)[-au]^n\,.$$

Looking at the various examples above and comparing them with the under-ing differential equations, we see immediately which difference approximations nat have been used and at which point they apply. Therefore, the compact otation effectively communicates the reasoning behind turning a differential quation into a difference equation.

Implementation

Goal.

We want make a computer program for solving

$$u'(t) = -au(t), \quad t \in (0,T], \quad u(0) = I,$$

by finite difference methods. The program should also display the num solution as a curve on the screen, preferably together with the solution.

All programs referred to in this section are found in the `src/decay`[1] d (we use the classical Unix term *directory* for what many others nowad *folder*).

Mathematical problem. We want to explore the Forward Euler sch Backward Euler, and the Crank-Nicolson schemes applied to our model From an implementational point of view, it is advantageous to implem θ-rule

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n,$$

since it can generate the three other schemes by various of choices of θ: θ Forward Euler, $\theta = 1$ for Backward Euler, and $\theta = 1/2$ for Crank-Nicolson a, $u^0 = I$, T, and Δt, our task is to use the θ-rule to compute $u^1, u^2,$ where $t_{N_t} = N_t\Delta t$, and N_t the closest integer to $T/\Delta t$.

Computer Language: Python. Any programming language can be generate the u^{n+1} values from the formula above. However, in this do we shall mainly make use of Python of several reasons:

- Python has a very clean, readable syntax (often known as "exe pseudo-code").
- Python code is very similar to MATLAB code (and MATLAI particularly widespread use for scientific computing).
- Python is a full-fledged, very powerful programming language.
- Python is similar to, but much simpler to work with and results reliable code than C++.
- Python has a rich set of modules for scientific computing, and its po in scientific computing is rapidly growing.
- Python was made for being combined with compiled languages (C Fortran) to reuse existing numerical software and to reach high c tional performance of new implementations.
- Python has extensive support for administrative task needed whe large-scale computational investigations.
- Python has extensive support for graphics (visualization, user in web applications).

[1] `http://tinyurl.com/jvzzcfn/decay`

- FEniCS, a very powerful tool for solving PDEs by the finite element method, is most human-efficient to operate from Python.

earning Python is easy. Many newcomers to the language will probably learn nough from the forthcoming examples to perform their own computer experiients. The examples start with simple Python code and gradually make use of nore powerful constructs as we proceed. As long as it is not inconvenient for ne problem at hand, our Python code is made as close as possible to MATLAB ode for easy transition between the two languages.

Readers who feel the Python examples are too hard to follow will probably enefit from reading a tutorial, e.g.,

- The Official Python Tutorial[2]
- Python Tutorial on tutorialspoint.com[3]
- Interactive Python tutorial site[4]
- A Beginner's Python Tutorial[5]

he author also has a comprehensive book [4] that teaches scientific programming ith Python from the ground up.

.1 Making a solver function

/e choose to have an array `u` for storing the u^n values, $n = 0, 1, \ldots, N_t$. The lgorithmic steps are

1. initialize u^0
2. for $t = t_n$, $n = 1, 2, \ldots, N_t$: compute u_n using the θ-rule formula

unction for computing the numerical solution. The following Python unction takes the input data of the problem (I, a, T, Δt, θ) as arguments and eturns two arrays with the solution $u^0, \ldots, u^{N_t}$ and the mesh points $t_0, \ldots, t_{N_t}$, espectively:

```
rom numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    Nt = int(T/dt)            # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh
```

[2] http://docs.python.org/2/tutorial/
[3] http://www.tutorialspoint.com/python/
[4] http://www.learnpython.org/
[5] http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial


```
    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

The `numpy` library contains a lot of functions for array computing of the function names are similar to what is found in the alternative s computing language MATLAB. Here we make use of

- `zeros(Nt+1)` for creating an array of a size `Nt+1` and initializ elements to zero
- `linspace(0, T, Nt+1)` for creating an array with `Nt+1` coordina formly distributed between `0` and `T`

The `for` loop deserves a comment, especially for newcomers to Pytho construction `range(0, Nt, s)` generates all integers from `0` to `Nt` in `s`, *but not including* `Nt`. Omitting `s` means `s=1`. For example, `range(0` gives `0` and `3`, while `range(0, Nt)` generates `0`, `1`, ..., `Nt-1`. Our loop the following assignments to `u[n+1]`: `u[1]`, `u[2]`, ..., `u[Nt]`, which is v want since `u` has length `Nt+1`. The first index in Python arrays or lists is `0` and the last is then `len(u)-1`. The length of an array `u` is obtained by or `u.size`.

To compute with the `solver` function, we need to *call* it. Here is a call:

```
u, t = solver(I=1, a=2, T=8, dt=0.8, theta=1)
```

Integer division. The shown implementation of the `solver` may face p and wrong results if `T`, `a`, `dt`, and `theta` are given as integers, see Exer and **??**. The problem is related to *integer division* in Python (as well as in C, C++, and many other computer languages): `1/2` becomes `0`, while `1/2.0`, or `1.0/2.0` all become `0.5`. It is enough that at least the nu or the denominator is a real number (i.e., a `float` object) to ensure mathematical division. Inserting a conversion `dt = float(dt)` guarant `dt` is `float` and avoids problems in Exercise **??**.

Another problem with computing $N_t = T/\Delta t$ is that we should roun the nearest integer. With `Nt = int(T/dt)` the `int` operation picks the integer smaller than `T/dt`. Correct mathematical rounding as known fro is obtained by

```
Nt = int(round(T/dt))
```

The complete version of our improved, safer `solver` function then beco

```
from numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

Doc strings. Right below the header line in the `solver` function there is a python string enclosed in triple double quotes `"""`. The purpose of this string object is to document what the function does and what the arguments are. In this case the necessary documentation do not span more than one line, but with triple double quoted strings the text may span several lines:

```
def solver(I, a, T, dt, theta):
    """
    Solve

        u'(t) = -a*u(t),

    with initial condition u(0)=I, for t in the time interval
    (0,T]. The time interval is divided into time steps of
    length dt.

    theta=1 corresponds to the Backward Euler scheme, theta=0
    to the Forward Euler scheme, and theta=0.5 to the Crank-
    Nicolson method.
    """
    ...
```

Such documentation strings appearing right after the header of a function are called *doc strings*. There are tools that can automatically produce nicely formatted documentation by extracting the definition of functions and the contents of doc strings.

It is strongly recommended to equip any function whose purpose is not obvious with a doc string. Nevertheless, the forthcoming text deviates from this rule if the function is explained in the text.

Formatting of numbers. Having computed the discrete solution `u`, it is natural to look at the numbers:

```
# Write out a table of t and u values:
for i in range(len(t)):
    print t[i], u[i]
```


This compact `print` statement gives unfortunately quite ugly output bec `t` and `u` values are not aligned in nicely formatted columns. To fix this p we recommend to use the *printf format*, supported most programming la inherited from C. Another choice is Python's recent *format string synt*

Writing `t[i]` and `u[i]` in two nicely formatted columns is done l with the printf format:

```
print 't=%6.3f u=%g' % (t[i], u[i])
```

The percentage signs signify "slots" in the text where the variables liste end of the statement are inserted. For each "slot" one must specify a for how the variable is going to appear in the string: `s` for pure text, `d` for an `g` for a real number written as compactly as possible, `9.3E` for scientific n with three decimals in a field of width 9 characters (e.g., `-1.351E-2`), or standard decimal notation with two decimals formatted with minimun The printf syntax provides a quick way of formatting tabular output of n with full control of the layout.

The alternative *format string syntax* looks like

```
print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```

As seen, this format allows logical names in the "slots" where `t[i]` and u to be inserted. The "slots" are surrounded by curly braces, and the logic is followed by a colon and then the printf-like specification of how to forn numbers, integers, or strings.

Running the program. The function and main program shown abo be placed in a file, say with name `decay_v1.py`[6] (`v1` for 1st version program). Make sure you write the code with a suitable text editor Emacs, Vim, Notepad++, or similar). The program is run by executing this way:

```
Terminal> python decay_v1.py
```

The text `Terminal>` just indicates a prompt in a Unix/Linux or DOS t window. After this prompt, which will look different in your terminal depending on the terminal application and how it is set up, comma `python decay_v1.py` can be issued. These commands are interpreted operating system.

We strongly recommend to run Python programs within the IPyth First start IPython by typing `ipython` in the terminal window. Ins IPython shell, our program `decay_v1.py` is run by the command `run dec`

[6] `http://tinyurl.com/jvzzcfn/decay/decay_v1.py`

```
erminal> ipython

n [1]: run decay_v1.py
= 0.000 u=1
= 0.800 u=0.384615
= 1.600 u=0.147929
= 2.400 u=0.0568958
= 3.200 u=0.021883
= 4.000 u=0.00841653
= 4.800 u=0.00323713
= 5.600 u=0.00124505
= 6.400 u=0.000478865
= 7.200 u=0.000184179
= 8.000 u=7.0838e-05

n [2]:
```

The advantage of running programs in IPython are many: previous commands re easily recalled with the up arrow, `%pdb` turns on debugging so that variables an be examined if the program aborts due to an exception, output of commands re stored in variables, programs and statements can be profiled, any operating ystem command can be executed, modules can be loaded automatically and ther customizations can be performed when starting IPython – to mention a w of the most useful features.

Although running programs in IPython is strongly recommended, most xecution examples in the forthcoming text use the standard Python shell with rompt >>> and run programs through a typesetting like

```
erminal> python programname
```

he reason is that such typesetting makes the text more compact in the vertical irection than showing sessions with IPython syntax.

lotting the solution. Having the `t` and `u` arrays, the approximate solution is visualized by the intuitive command `plot(t, u)`:

```
rom matplotlib.pyplot import *
lot(t, u)
how()
```

will be illustrative to also plot $u_e(t)$ for comparison. We first need to make function for computing the analytical solution $u_e(t) = Ie^{-at}$ of the model roblem:

```
ef exact_solution(t, I, a):
    return I*exp(-a*t)
```

is tempting to just do


```
u_e = exact_solution(t, I, a)
plot(t, u, t, u_e)
```

However, this is not exactly what we want: the `plot` function draws straig between the discrete points `(t[n], u_e[n])` while $u_e(t)$ varies as an exp function between the mesh points. The technique for showing the variation of $u_e(t)$ between the mesh points is to introduce a very fine n $u_e(t)$:

```
t_e = linspace(0, T, 1001)      # fine mesh
u_e = exact_solution(t_e, I, a)
```

We can also plot the curves with different colors and styles, e.g.,

```
plot(t_e, u_e, 'b-',          # blue line for u_e
     t,   u,   'r--o')        # red dashes w/circles
```

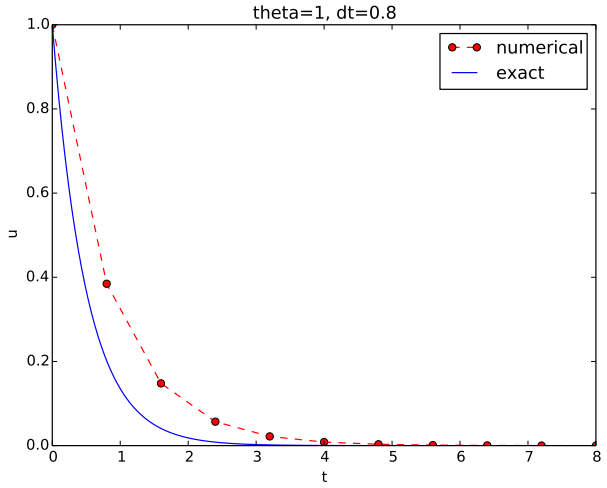
With more than one curve in the plot we need to associate eac with a legend. We also want appropriate names on the axis, a title, an containing the plot as an image for inclusion in reports. The Matplotlib (`matplotlib.pyplot`) contains functions for this purpose. The name functions are similar to the plotting functions known from MATLAB. A c function for creating the comparison plot becomes

```
from matplotlib.pyplot import *

def plot_numerical_and_exact(theta, I, a, T, dt):
    """Compare the numerical and exact solution in a plot."""
    u, t = solver(I=I, a=a, T=T, dt=dt, theta=theta)

    t_e = linspace(0, T, 1001)        # fine mesh for u_e
    u_e = exact_solution(t_e, I, a)

    plot(t,   u,   'r--o',            # red dashes w/circles
         t_e, u_e, 'b-')              # blue line for exact sol.
    legend(['numerical', 'exact'])
    xlabel('t')
    ylabel('u')
    title('theta=%g, dt=%g' % (theta, dt))
    savefig('plot_%s_%g.png' % (theta, dt))

plot_numerical_and_exact(I=1, a=2, T=8, dt=0.8, theta=1)
show()
```

Note that `savefig` here creates a PNG file whose name reflects the val and Δt so that we can easily distinguish files from different runs with θ

The complete code is found in the file `decay_v2.py`[7]. The resulting shown in Figure **??**. As seen, there is quite some discrepancy between th and the numerical solution. Fortunately, the numerical solution approac exact one as Δt is reduced.

[7] `http://tinyurl.com/jvzzcfn/decay/decay_v2.py`



.2 Verifying the implementation

; is easy to make mistakes while deriving and implementing numerical algo-
thms, so we should never believe in the solution before it has been thoroughly
erified. The most obvious idea to verify the computations is to compare the
umerical solution with the exact solution, when that exists, but there will
lways be a discrepancy between these two solutions because of the numerical
pproximations. The challenging question is whether we have the mathematically
orrect discrepancy or if we have another, maybe small, discrepancy due to both
n approximation error and an error in the implementation. When looking at
igure **??**, it is impossible to judge whether the program is correct or not.

The purpose of *verifying* a program is to bring evidence for the property
iat there are no errors in the implementation. To avoid mixing unavoidable
pproximation errors and undesired implementation errors, we should try to
iake tests where we have some exact computation of the discrete solution or at
ast parts of it. Examples will show how this can be done.

unning a few algorithmic steps by hand. The simplest approach to
roduce a correct reference for the discrete solution u of finite difference equations
to compute a few steps of the algorithm by hand. Then we can compare the
and calculations with numbers produced by the program.

A straightforward approach is to use a calculator and compute u^1, u^2, and
^{3}. With $I = 0.1$, $\theta = 0.8$, and $\Delta t = 0.8$ we get

$$A \equiv \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t} = 0.298245614035$$

$$u^1 = AI = 0.0298245614035,$$
$$u^2 = Au^1 = 0.00889504462912,$$
$$u^3 = Au^2 = 0.00265290804728$$

Comparison of these manual calculations with the result of the
function is carried out in the function

```
def test_solver_three_steps():
    """Compare three steps with known manual computations."""
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    u_by_hand = array([I,
                       0.0298245614035,
                       0.00889504462912,
                       0.00265290804728])

    Nt = 3  # number of time steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)

    tol = 1E-15  # tolerance for comparing floats
    diff = abs(u - u_by_hand).max()
    success = diff <= tol
    assert success
```

The `test_solver_three_steps` function follows widely used convent
unit testing. By following such conventions we can at a later stage easily
a big test suite for our software. The conventions are three-fold:

- The test function starts with `test_` and takes no arguments.
- The test ends up in a boolean expression that is `True` if the test
and `False` if it failed.
- The function runs `assert` on the boolean expression, resulting in
abortion (due to an `AssertionError` exception) if the test failed.

The main program can routinely run the verification test prior to solv
real problem:

```
test_solver_three_steps()
plot_numerical_and_exact(I=1, a=2, T=8, dt=0.8, theta=1)
show()
```

(Rather than calling `test_*()` functions explicitly, one will normally ask
framework like nose or pytest to find and run such functions.) The c
program including the verification above is found in the file `decay_v3.`

2.3 Computing the numerical error as a mesh funct

Now that we have some evidence for a correct implementation, we
position to compare the computed u^n values in the `u` array with the
values at the mesh points, in order to study the error in the numerical

[8] `http://tinyurl.com/jvzzcfn/decay/decay_v3.py`

A natural way to compare the exact and discrete solutions is to calculate ıeir difference as a mesh function:

$$e^n = u_{\mathrm{e}}(t_n) - u^n, \quad n = 0, 1, \ldots, N_t\,. \tag{34}$$

Ve may view $u^n_{\mathrm{e}} = u_{\mathrm{e}}(t_n)$ as the representation of $u_{\mathrm{e}}(t)$ as a mesh function ıther than a continuous function defined for all $t \in [0, T]$ (u^n_{e} is often called the *epresentative* of u_{e} on the mesh). Then, $e^n = u^n_{\mathrm{e}} - u^n$ is clearly the difference f two mesh functions. This interpretation of e^n is natural when programming.

The error mesh function e^n can be computed by

```
ı, t = solver(I, a, T, dt, theta)  # Numerical sol.
ı_e = exact_solution(t, I, a)      # Representative of exact sol.
e = u_e - u
```

ote that the mesh functions `u` and `u_e` are represented by arrays and associated ith the points in the array `t`.

Array arithmetics.

The last statements

```
u_e = exact_solution(t, I, a)
e = u_e - u
```

are primary examples of array arithmetics: `t` is an array of mesh points that we pass to `exact_solution`. This function evaluates `-a*t`, which is a scalar times an array, meaning that the scalar is multiplied with each array element. The result is an array, let us call it `tmp1`. Then `exp(tmp1)` means applying the exponential function to each element in `tmp`, resulting an array, say `tmp2`. Finally, `I*tmp2` is computed (scalar times array) and `u_e` refers to this array returned from `exact_solution`. The expression `u_e - u` is the difference between two arrays, resulting in a new array referred to by `e`.

.4 Computing the norm of the numerical error

ıstead of working with the error e^n on the entire mesh, we often want one umber expressing the size of the error. This is obtained by taking the norm of ıe error function.

Let us first define norms of a function $f(t)$ defined for all $t \in [0, T]$. Three ɔmmon norms are

$$||f||_{L^2} = \left(\int_0^T f(t)^2 dt\right)^{1/2},$$
$$||f||_{L^1} = \int_0^T |f(t)| dt,$$
$$||f||_{L^\infty} = \max_{t\in[0,T]} |f(t)|\,.$$

The L^2 norm (35) ("L-two norm") has nice mathematical properties an most popular norm. It is a generalization of the well-known Eucledia of vectors to functions. The L^∞ is also called the max norm or the su norm. In fact, there is a whole family of norms,

$$||f||_{L^p} = \left(\int_0^T f(t)^p dt\right)^{1/p},$$

with p real. In particular, $p = 1$ corresponds to the L^1 norm above whil is the L^∞ norm.

Numerical computations involving mesh functions need correspondin Given a set of function values, f^n, and some associated mesh point numerical integration rule can be used to calculate the L^2 and L^1 norms above. Imagining that the mesh function is extended to vary linearly l the mesh points, the Trapezoidal rule is in fact an exact integration possible modification of the L^2 norm for a mesh function f^n on a unifor with spacing Δt is therefore the well-known Trapezoidal integration for

$$||f^n|| = \left(\Delta t\left(\frac{1}{2}(f^0)^2 + \frac{1}{2}(f^{N_t})^2 + \sum_{n=1}^{N_t-1}(f^n)^2\right)\right)^{1/2}$$

A common approximation of this expression, motivated by the conven having a simpler formula, is

$$||f^n||_{\ell^2} = \left(\Delta t\sum_{n=0}^{N_t}(f^n)^2\right)^{1/2}.$$

This is called the discrete L^2 norm and denoted by ℓ^2. The error i compared with the Trapezoidal integration formula is $\Delta t((f^0)^2 + (f$ which means perturbed weights at the end points of the mesh function, error goes to zero as $\Delta t \to 0$. As long as we are consistent and stick to c of integration rule for the norm of a mesh function, the details and acc this rule is not of concern.

The three discrete norms for a mesh function f^n, corresponding to L^1, and L^∞ norms of $f(t)$ defined above, are defined by

$$||f^n||_{\ell^2}\left(\Delta t\sum_{n=0}^{N_t}(f^n)^2\right)^{1/2}, \tag{39}$$

$$||f^n||_{\ell^1}\Delta t\sum_{n=0}^{N_t}|f^n| \tag{40}$$

$$||f^n||_{\ell^\infty}\max_{0\leq n\leq N_t}|f^n|\,. \tag{41}$$

Note that the L^2, L^1, ℓ^2, and ℓ^1 norms depend on the length of the interval f interest (think of $f=1$, then the norms are proportional to $\sqrt{T}$ or T). In ome applications it is convenient to think of a mesh function as just a vector of unction values and neglect the information of the mesh points. Then we can eplace Δt by T/N_t and drop T. Moreover, it is convenient to divide by the otal length of the vector, N_t+1, instead of N_t. This reasoning gives rise to the *ector norms* for a vector $f=(f_0,\ldots,f_N)$:

$$||f||_2=\left(\frac{1}{N+1}\sum_{n=0}^{N}(f_n)^2\right)^{1/2}, \tag{42}$$

$$||f||_1=\frac{1}{N+1}\sum_{n=0}^{N}|f_n| \tag{43}$$

$$||f||_{\ell^\infty}=\max_{0\leq n\leq N}|f_n|\,. \tag{44}$$

ere we have used the common vector component notation with subscripts (f_n) nd N as length. We will mostly work with mesh functions and use the discrete ² norm (39) or the max norm ℓ^∞ (41), but the corresponding vector norms 2)-(44) are also much used in numerical computations, so it is important to now the different norms and the relations between them.

A single number that expresses the size of the numerical error will be taken s $||e^n||_{\ell^2}$ and called E:

$$E=\sqrt{\Delta t\sum_{n=0}^{N_t}(e^n)^2} \tag{45}$$

he corresponding Python code, using array arithmetics, reads

```
E = sqrt(dt*sum(e**2))
```

he `sum` function comes from `numpy` and computes the sum of the elements of n array. Also the `sqrt` function is from `numpy` and computes the square root of ach element in the array argument.

Scalar computing. Instead of doing array computing `sqrt(dt*sum` we can compute with one element at a time:

```
m = len(u)     # length of u array (alt: u.size)
u_e = zeros(m)
t = 0
for i in range(m):
    u_e[i] = exact_solution(t, a, I)
    t = t + dt
e = zeros(m)
for i in range(m):
    e[i] = u_e[i] - u[i]
s = 0  # summation variable
for i in range(m):
    s = s + e[i]**2
error = sqrt(dt*s)
```

Such element-wise computing, often called *scalar* computing, takes mo is less readable, and runs much slower than what we can achieve wit computing.

2.5 Plotting solutions

2.6 Experiments with computing and plotting

Let us wrap up the computation of the error measure and all the p statements for comparing the exact and numerical solution in a new f `explore`. This function can be called for various θ and Δt values to see error varies with the method and the mesh resolution:

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True)
    """
    u, t = solver(I, a, T, dt, theta)    # Numerical solution
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*sum(e**2))
    if makeplot:
        figure()                         # create new plot
        t_e = linspace(0, T, 1001)       # fine mesh for u_e
        u_e = exact_solution(t_e, I, a)
        plot(t,   u,   'r--o')           # red dashes w/circles
        plot(t_e, u_e, 'b-')             # blue line for exact so
        legend(['numerical', 'exact'])
        xlabel('t')
        ylabel('u')
        title('theta=%g, dt=%g' % (theta, dt))
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        savefig('%s_%g.png' % (theta2name[theta], dt))
        savefig('%s_%g.pdf' % (theta2name[theta], dt))
        show()
    return E
```

The `figure()` call is key: without it, a new `plot` command will draw the ew pair of curves in the same plot window, while we want the different pairs to ppear in separate windows and files. Calling `figure()` ensures this.

Filenames with the method name (FE, BE, or CN) rather than the θ value mbedded in the name, can easily be created with the aid of a little Python ictionary for mapping θ to method acronyms:

```
theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
savefig('%s_%g.png' % (theta2name[theta], dt))
```

The `explore` function stores the plot in two different image file formats: NG and PDF. The PNG format is aimed at being included in HTML files and ie PDF format in LaTeX documents (more precisely, in pdfLaTeX documents). requently used viewers for these image files on Unix systems are `gv` (comes ith Ghostscript) for the PDF format and `display` (from the ImageMagick) uite for PNG files:

```
erminal> gv BE_0.5.pdf
erminal> display BE_0.5.png
```

A main program may run a loop over the three methods (θ values) and call xplore to compute errors and make plots:

```
def main(I, a, T, dt_values, theta_values=(0, 0.5, 1)):
    for theta in theta_values:
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot=True)
            print '%3.1f %6.2f: %12.3E' % (theta, dt, E)
```

he complete code containing the functions above resides in the file `decay_ lot_mpl.py`[9]. Running this program results in

```
erminal> python decay_plot_mpl.py
.0   0.40:     2.105E-01
.0   0.04:     1.449E-02
.5   0.40:     3.362E-02
.5   0.04:     1.887E-04
.0   0.40:     1.030E-01
.0   0.04:     1.382E-02
```

/e observe that reducing Δt by a factor of 10 increases the accuracy for all ree methods (θ values). We also see that the combination of $\theta = 0.5$ and a nall time step $\Delta t = 0.04$ gives a much more accurate solution, and that $\theta = 0$ nd $\theta = 1$ with $\Delta t = 0.4$ result in the least accurate solutions.

Figure 6 demonstrates that the numerical solution for $\Delta t = 0.4$ clearly lies elow the exact curve, but that the accuracy improves considerably by reducing ie time step by a factor of 10.

[9] `http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py`

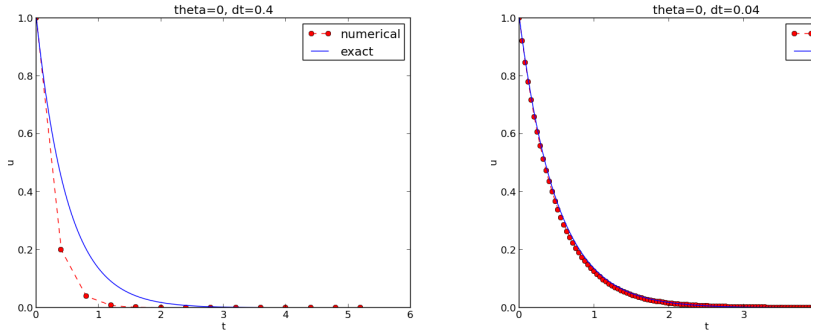


Figure 6: The Forward Euler scheme for two values of the time st

Combining plot files. Mounting two PNG files, as done in the figure, done by the `montage`[10] program from the ImageMagick suite:

```
Terminal> montage -background white -geometry 100% -tile 2x1 \
          FE_0.4.png FE_0.04.png FE1.png
Terminal> convert -trim FE1.png FE1.png
```

The `-geometry` argument is used to specify the size of the image, and preserve the individual sizes of the images. The `-tile HxV` option sp images in the horizontal direction and `V` images in the vertical direction. of image files to be combined are then listed, with the name of the r combined image, here `FE1.png` at the end. The `convert -trim` co removes surrounding white areas in the figure (an operation usually k *cropping* in image manipulation programs).

For LaTeX reports it is not recommended to use `montage` and PNG fil result has too low resolution. Instead, plots should be made in the PDF and combined using the `pdftk`, `pdfnup`, and `pdfcrop` tools (on Linux/

```
Terminal> pdftk FE_0.4.png FE_0.04.png output tmp.pdf
Terminal> pdfnup --nup 2x1 --outfile tmp.pdf tmp.pdf
Terminal> pdfcrop tmp.pdf FE1.png  # output in FE1.png
```

Here, `pdftk` combines images into a multi-page PDF file, `pdfnup` comb images in individual pages to a table of images (pages), and `pdfcrop` white margins in the resulting combined image file.

The behavior of the two other schemes is shown in Figures 7 and 8. Nicolson is obviously the most accurate scheme from this visual point

[10] `http://www.imagemagick.org/script/montage.php`

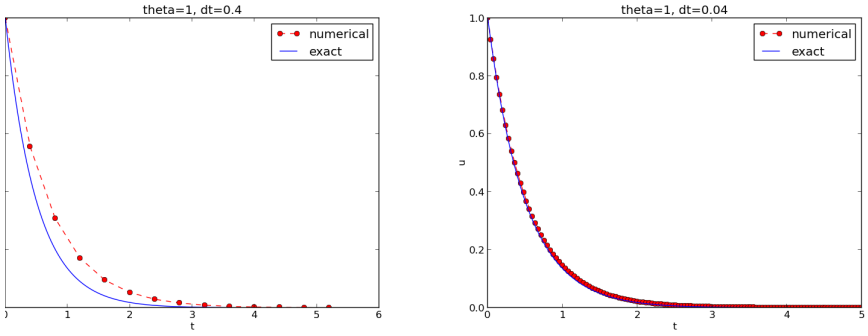


Figure 7: The Backward Euler scheme for two values of the time step.

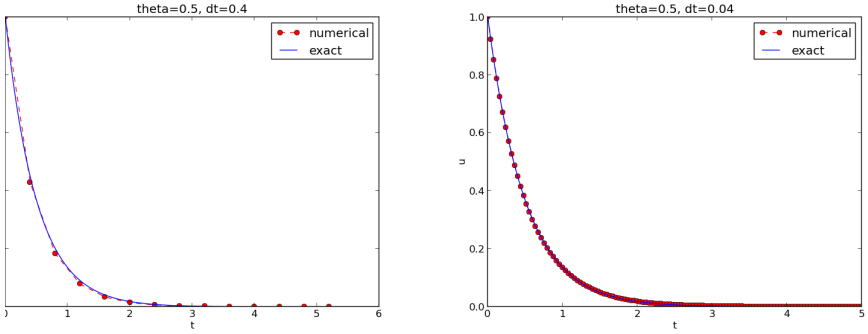


Figure 8: The Crank-Nicolson scheme for two values of the time step.

'lotting with SciTools. The SciTools package[11] provides a unified plotting ıterface, called Easyviz, to many different plotting packages, including Matlotlib, Gnuplot, Grace, MATLAB, VTK, OpenDX, and VisIt. The syntax is ery similar to that of Matplotlib and MATLAB. In fact, the plotting commands ıown above look the same in SciTool's Easyviz interface, apart from the import atement, which reads

```
rom scitools.std import *
```

his statement performs a `from numpy import *` as well as an import of the ıost common pieces of the Easyviz (`scitools.easyviz`) package, along with ome additional numerical functionality.

With Easyviz one can merge several plotting commands into a single one sing keyword arguments:

[11] `http://code.google.com/p/scitools`


```
plot(t,   u,   'r--o',          # red dashes w/circles
     t_e, u_e, 'b-',            # blue line for exact sol.
     legend=['numerical', 'exact'],
     xlabel='t',
     ylabel='u',
     title='theta=%g, dt=%g' % (theta, dt),
     savefig='%s_%g.png' % (theta2name[theta], dt),
     show=True)
```

The `decay_plot_st.py`[12] file contains such a demo.

By default, Easyviz employs Matplotlib for plotting, but Gnuplo Grace[14] are viable alternatives:

```
Terminal> python decay_plot_st.py --SCITOOLS_easyviz_backend gnupl
Terminal> python decay_plot_st.py --SCITOOLS_easyviz_backend grace
```

The backend used for creating plots (and numerous other options) permanently set in SciTool's configuration file.

All the Gnuplot windows are launched without any need to kill on the next one pops up (as is the case with Matplotlib) and one can press 'q' anywhere in a plot window to kill it. Another advantage of Gnuplo automatic choice of sensible and distinguishable line types in black-an PDF and PostScript files.

Regarding functionality for annotating plots with title, labels on t legends, etc., we refer to the documentation of Matplotlib and SciTools f detailed information on the syntax. The hope is that the programming explained so far suffices for understanding the code and learning more combination of the forthcoming examples and other resources such as bo web pages.

Test the understanding.

Exercise 11 asks you to implement a solver for a problem that is sli different from the one above. You may use the `solver` and `exp` functions explained above as a starting point. Apply the new solv Exercise 12.

2.7 Memory-saving implementation

The computer memory requirements of our implementations so far mainly of the `u` and `t` arrays, both of length $N_t + 1$, plus some other te arrays that Python needs for intermediate results if we do array arithm

[12] `http://tinyurl.com/jvzzcfn/decay/decay_plot_st.py`
[13] `http://www.gnuplot.info/`
[14] `http://plasma-gate.weizmann.ac.il/Grace/`

ur program (e.g., `I*exp(-a*t)` needs to store `a*t` before `-` can be applied to it nd then `exp`). Regardless of how we implement simple ODE problems, storage equirements are very modest and put not restriction on how we choose our data tructures and algorithms. Nevertheless, when the methods for ODEs used here re applied to three-dimensional partial differential equation (PDE) problems, emory storage requirements suddenly become a challenging issue.

The PDE counterpart to our model problem $u' = -a$ is a diffusion equation $_t = a\nabla^2 u$ posed on a space-time domain. The discrete representation of this omain may in 3D be a spatial mesh of M^3 points and a time mesh of N_t oints. A typical desired value for M is 100 in many applications, or even 1000. toring all the computed u values, like we have done in the programs so far, emands storage of some arrays of size $M^3 N_t$, giving a factor of M^3 larger orage demands compared to our ODE programs. Each real number in the rray for u requires 8 bytes (b) of storage. With $M = 100$ and $N_t = 1000$, here is a storage demand of $(10^3)^3 \cdot 1000 \cdot 8 = 8$ Gb for the solution array. ortunately, we can usually get rid of the N_t factor, resulting in 8 Mb of storage. elow we explain how this is done, and the technique is almost always applied implementations of PDE problems.

Let us critically evaluate how much we really need to store in the computer's emory in our implementation of the θ method. To compute a new u^{n+1}, all we eed is u^n. This implies that the previous $u^{n-1}, u^{n-2}, \ldots, u^0$ values do not need be stored in an array, although this is convenient for plotting and data analysis the program. Instead of the `u` array we can work with two variables for real umbers, `u` and `u_1`, representing u^{n+1} and u^n in the algorithm, respectively. t each time level, we update `u` from `u_1` and then set `u_1 = u` so that the omputed u^{n+1} value becomes the "previous" value u^n at the next time level. he downside is that we cannot plot the solution after the simulation is done nce only the last two numbers are available. The remedy is to store computed alues in a file and use the file for visualizing the solution later.

We have implemented this memory saving idea in the file `decay_memsave.y`[15], which is a slight modification of `decay_plot_mpl.py`[16] program.

The following function demonstrates how we work with the two most recent alues of the unknown:

```
def solver_memsave(I, a, T, dt, theta, filename='sol.dat'):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.
    Minimum use of memory. The solution is stored in a file
    (with name filename) for later plotting.
    """
    dt = float(dt)         # avoid integer division
    Nt = int(round(T/dt))  # no of intervals

    outfile = open(filename, 'w')
    # u: time level n+1, u_1: time level n
    t = 0
```

[15] http://tinyurl.com/jvzzcfn/decay/decay_memsave.py
[16] http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py


```
    u_1 = I
    outfile.write('%.16E  %.16E\n' % (t, u_1))
    for n in range(1, Nt+1):
        u = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u_1
        u_1 = u
        t += dt
        outfile.write('%.16E  %.16E\n' % (t, u))
    outfile.close()
    return u, t
```

This code snippet serves as a quick introduction to file writing in Python. the data in the file into arrays `t` and `u` are done by the function

```
def read_file(filename='sol.dat'):
    infile = open(filename, 'r')
    u = [];  t = []
    for line in infile:
        words = line.split()
        if len(words) != 2:
            print 'Found more than two numbers on a line!', words
            sys.exit(1)  # abort
        t.append(float(words[0]))
        u.append(float(words[1]))
    return np.array(t), np.array(u)
```

This type of file with numbers in rows and columns is very common, an has a function `loadtxt` which loads such tabular data into a two-dim array, say with name `data`. The number in row `i` and column `j` is then `dat` The whole column number `j` can be extracted by `data[:,j]`. A ve `read_file` using `np.loadtxt` reads

```
def read_file_numpy(filename='sol.dat'):
    data = np.loadtxt(filename)
    t = data[:,0]
    u = data[:,1]
    return t, u
```

The present counterpart to the `explore` function from `decay_plot_m` must run `solver_memsave` and then load data from file before we can c the error measure and make the plot:

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    filename = 'u.dat'
    u, t = solver_memsave(I, a, T, dt, theta, filename)

    t, u = read_file(filename)
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*np.sum(e**2))
    if makeplot:
        figure()
        ...
```

[17] http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py

Apart from the internal implementation, where u^n values are stored in file rather than in an array, `decay_memsave.py` file works exactly as the `ecay_plot_mpl.py` file.

Analysis of finite difference equations

/e address the ODE for exponential decay,

$$u'(t) = -au(t), \quad u(0) = I, \tag{46}$$

here a and I are given constants. This problem is solved by the θ-rule finite ifference scheme, resulting in the recursive equations

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n \tag{47}$$

r the numerical solution u^{n+1}, which approximates the exact solution u_e at time oint t_{n+1}. For constant mesh spacing, which we assume here, $t_{n+1} = (n+1)\Delta t$.

)iscouraging numerical solutions. Choosing $I = 1$, $a = 2$, and running xperiments with $\theta = 1, 0.5, 0$ for $\Delta t = 1.25, 0.75, 0.5, 0.1$, gives the results in igures 9, 10, and 11.

The characteristics of the displayed curves can be summarized as follows:

- The Backward Euler scheme always gives a monotone solution, lying above the exact curve.
- The Crank-Nicolson scheme gives the most accurate results, but for $\Delta t = 1.25$ the solution oscillates.
- The Forward Euler scheme gives a growing, oscillating solution for $\Delta t = 1.25$; a decaying, oscillating solution for $\Delta t = 0.75$; a strange solution $u^n = 0$ for $n \geq 1$ when $\Delta t = 0.5$; and a solution seemingly as accurate as the one by the Backward Euler scheme for $\Delta t = 0.1$, but the curve lies below the exact solution.

ince the exact solution of our model problem is a monotone function, $u(t) = e^{-at}$, some of these qualitatively wrong results are indeed alarming!

Goal.

We ask the question

- Under what circumstances, i.e., values of the input data I, a, and Δt will the Forward Euler and Crank-Nicolson schemes result in undesired oscillatory solutions?

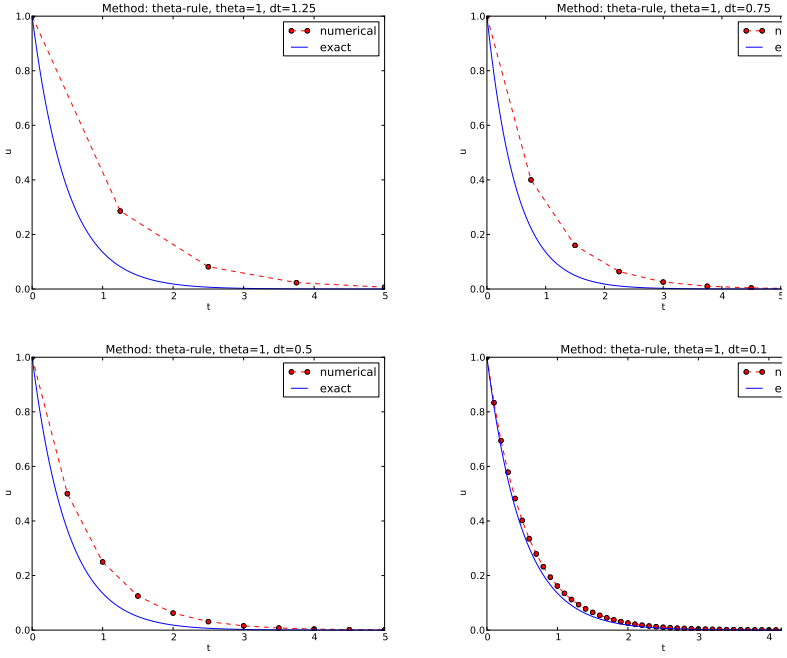


Figure 9: Backward Euler.

The question will be investigated both by numerical experiments ar precise mathematical theory. The latter will help establish general cr on Δt for avoiding non-physical oscillatory or growing solutions.

Another question to be raised is

- How does Δt impact the error in the numerical solution?

For our simple model problem we can answer this question very prec but we will also look at simplified formulas for small Δt and touch important concepts such as *convergence rate* and *the order of a sc* Other fundamental concepts mentioned are stability, consistency, convergence.

3.1 Experimental investigation of oscillatory solutic

To address the first question above, we may set up an experiment w loop over values of I, a, and Δt. For each experiment, we flag the sol oscillatory if

$$u^n > u^{n-1},$$

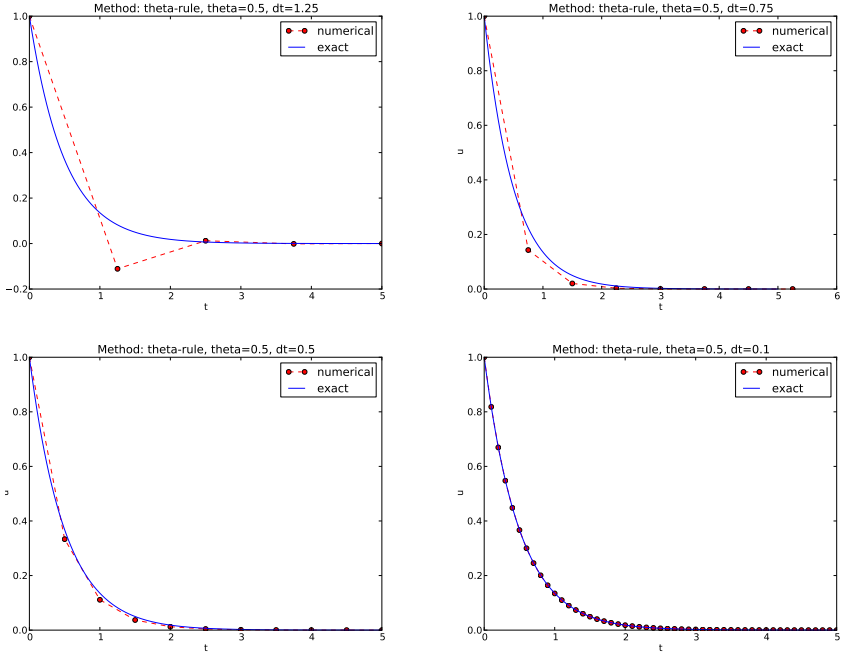


Figure 10: Crank-Nicolson.

or some value of n, since we expect u^n to decay with n, but oscillations make u increase over a time step. We will quickly see that oscillations are independent of I, but do depend on a and Δt. Therefore, we introduce a two-dimensional function $B(a, \Delta t)$ which is 1 if oscillations occur and 0 otherwise. We can visualize B as a contour plot (lines for which $B = \text{const}$). The contour $B = 0.5$ corresponds to the borderline between oscillatory regions with $B = 1$ and monotone regions with $B = 0$ in the $a, \Delta t$ plane.

The B function is defined at discrete a and Δt values. Say we have given P a values, $a_0, \ldots, a_{P-1}$, and Q Δt values, $\Delta t_0, \ldots, \Delta t_{Q-1}$. These a_i and Δt_j values, $i = 0, \ldots, P-1$, $j = 0, \ldots, Q-1$, form a rectangular mesh of $P \times Q$ points in the plane. At each point $(a_i, \Delta t_j)$, we associate the corresponding value of $B(a_i, \Delta t_j)$, denoted B_{ij}. The B_{ij} values are naturally stored in a two-dimensional array. We can thereafter create a plot of the contour line $B_{ij} = 0.5$ dividing the oscillatory and monotone regions. The file `decay_osc_regions.py`[18] `osc_regions` stands for "oscillatory regions") contains all nuts and bolts to produce the $B = 0.5$ line in Figures 12 and 13. The oscillatory region is above this line.

[18] `http://tinyurl.com/jvzzcfn/decay/decay_osc_regions.py`

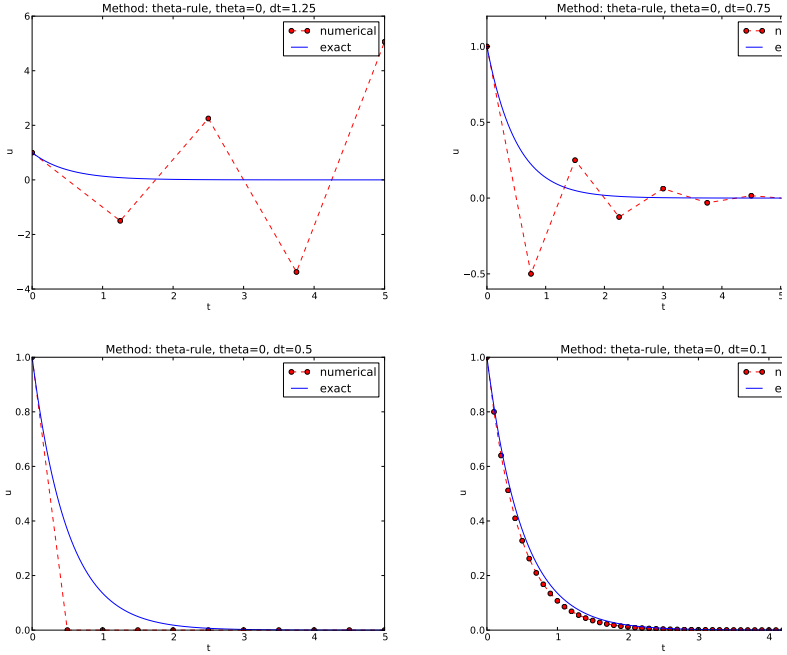
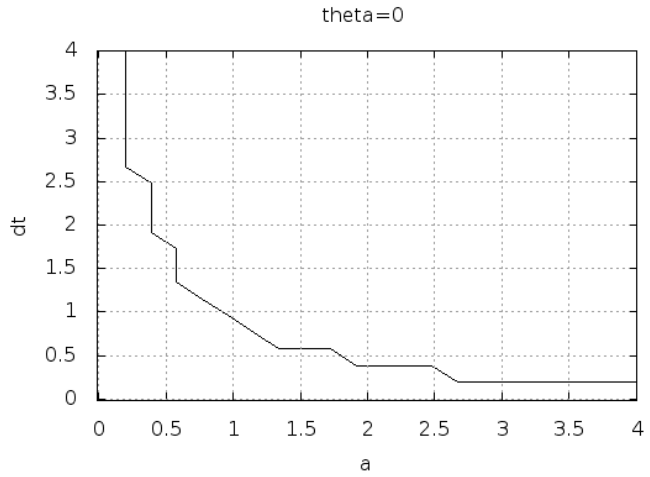


Figure 11: Forward Euler.

```
from decay_mod import solver
import numpy as np
import scitools.std as st

def non_physical_behavior(I, a, T, dt, theta):
    """
    Given lists/arrays a and dt, and numbers I, dt, and theta,
    make a two-dimensional contour line B=0.5, where B=1>0.5
    means oscillatory (unstable) solution, and B=0<0.5 means
    monotone solution of u'=-au.
    """
    a = np.asarray(a); dt = np.asarray(dt)  # must be arrays
    B = np.zeros((len(a), len(dt)))         # results
    for i in range(len(a)):
        for j in range(len(dt)):
            u, t = solver(I, a[i], T, dt[j], theta)
            # Does u have the right monotone decay properties?
            correct_qualitative_behavior = True
            for n in range(1, len(u)):
                if u[n] > u[n-1]:  # Not decaying?
                    correct_qualitative_behavior = False
                    break  # Jump out of loop
            B[i,j] = float(correct_qualitative_behavior)
    a_, dt_ = st.ndgrid(a, dt)  # make mesh of a and dt values
    st.contour(a_, dt_, B, 1)
    st.grid('on')
```

```
    st.title('theta=%g' % theta)
    st.xlabel('a'); st.ylabel('dt')
    st.savefig('osc_region_theta_%s.png' % theta)
    st.savefig('osc_region_theta_%s.pdf' % theta)

non_physical_behavior(
    I=1,
    a=np.linspace(0.01, 4, 22),
    dt=np.linspace(0.01, 4, 22),
    T=6,
    theta=0.5)
```



igure 12: Forward Euler scheme: oscillatory solutions occur for points above ıe curve.

By looking at the curves in the figures one may guess that $a\Delta t$ must be less ıan a critical limit to avoid the undesired oscillations. This limit seems to be bout 2 for Crank-Nicolson and 1 for Forward Euler. We shall now establish precise mathematical analysis of the discrete model that can explain the bservations in our numerical experiments.

.2 Exact numerical solution

tarting with $u^0 = I$, the simple recursion (47) can be applied repeatedly n mes, with the result that

$$u^n = IA^n, \quad A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}. \tag{48}$$

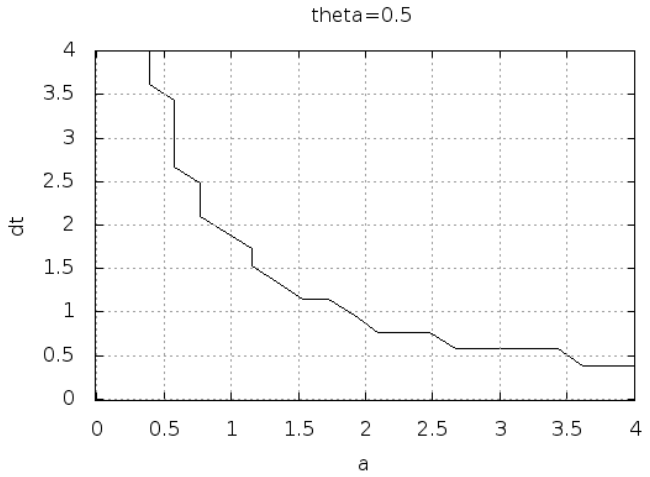


Figure 13: Crank-Nicolson scheme: oscillatory solutions occur for point the curve.

Solving difference equations.

Difference equations where all terms are linear in u^{n+1}, u^n, and m u^{n-1}, u^{n-2}, etc., are called *homogeneous, linear* difference equations their solutions are generally of the form $u^n = A^n$. Inserting this expre and dividing by A^{n+1} gives a polynomial equation in A. In the pr case we get

$$A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}.$$

This is a solution technique of wider applicability than repeated use c recursion (47).

Regardless of the solution approach, we have obtained a formula for ı formula can explain everything what we see in the figures above, but gives us a more general insight into accuracy and stability properties of t schemes.

3.3 Stability

Since u^n is a factor A raised to an integer power n, we realize that A for odd powers imply $u^n < 0$ and for even power result in $u^n > 0$. Tha

olution oscillates between the mesh points. We have oscillations due to $A < 0$ hen

$$(1-\theta)a\Delta t > 1\,. \tag{49}$$

ince $A > 0$ is a requirement for having a numerical solution with the same asic property (monotonicity) as the exact solution, we may say that $A > 0$ is a *ability criterion*. Expressed in terms of Δt the stability criterion reads

$$\Delta t < \frac{1}{(1-\theta)a}\,. \tag{50}$$

The Backward Euler scheme is always stable since $A < 0$ is impossible for $= 1$, while non-oscillating solutions for Forward Euler and Crank-Nicolson emand $\Delta t \leq 1/a$ and $\Delta t \leq 2/a$, respectively. The relation between Δt and a ook reasonable: a larger a means faster decay and hence a need for smaller time eps.

Looking at Figure 11, we see that with $a\Delta t = 2 \cdot 1.25 = 2.5$, $A = -1.5$, and ne solution $u^n = (-1.5)^n$ oscillates *and* grows. With $a\Delta t = 2 \cdot 0.75 = 1.5$, $= -0.5$, $u^n = (-0.5)^n$ decays but oscillates. The peculiar case $\Delta t = 0.5$, here the Forward Euler scheme produces a solution that is stuck on the t axis, orresponds to $A = 0$ and therefore $u^0 = I = 1$ and $u^n = 0$ for $n \geq 1$. The ecaying oscillations in the Crank-Nicolson scheme for $\Delta t = 1.25$ are easily xplained by the fact that $A \approx -0.11 < 0$.

The factor A is called the *amplification factor* since the solution at a new time evel is A times the solution at the previous time level. For a decay process, we ust obviously have $|A| \leq 1$, which is fulfilled for all Δt if $\theta \geq 1/2$. Arbitrarily arge values of u can be generated when $|A| > 1$ and n is large enough. The umerical solution is in such cases totally irrelevant to an ODE modeling decay rocesses! To avoid this situation, we must for $\theta < 1/2$ have

$$\Delta t \leq \frac{2}{(1-2\theta)a}, \tag{51}$$

hich means $\Delta t < 2/a$ for the Forward Euler scheme.

Stability properties.

We may summarize the stability investigations as follows:

1. The Forward Euler method is a *conditionally stable* scheme because it requires $\Delta t < 2/a$ for avoiding growing solutions and $\Delta t < 1/a$ for avoiding oscillatory solutions.

2. The Crank-Nicolson is *unconditionally stable* with respect to growing solutions, while it is conditionally stable with the criterion $\Delta t < 2/a$ for avoiding oscillatory solutions.

3. The Backward Euler method is unconditionally stable with re to growing and oscillatory solutions - any Δt will work.

Much literature on ODEs speaks about L-stable and A-stable method our case A-stable methods ensures non-growing solutions, while L-s methods also avoids oscillatory solutions.

3.4 Comparing amplification factors

After establishing how A impacts the qualitative features of the solution, now look more into how well the numerical amplification factor appro the exact one. The exact solution reads $u(t) = Ie^{-at}$, which can be rew

$$u_{\mathrm{e}}(t_n) = Ie^{-an\Delta t} = I(e^{-a\Delta t})^n\,.$$

From this formula we see that the exact amplification factor is

$$A_{\mathrm{e}} = e^{-a\Delta t}\,.$$

We realize that the exact and numerical amplification factors depend o Δt through the product $a\Delta t$. Therefore, it is convenient to introduce a for this product, $p = a\Delta t$, and view A and A_{e} as functions of p. Figure 1 these functions. Crank-Nicolson is clearly closest to the exact ampli factor, but that method has the unfortunate oscillatory behavior when

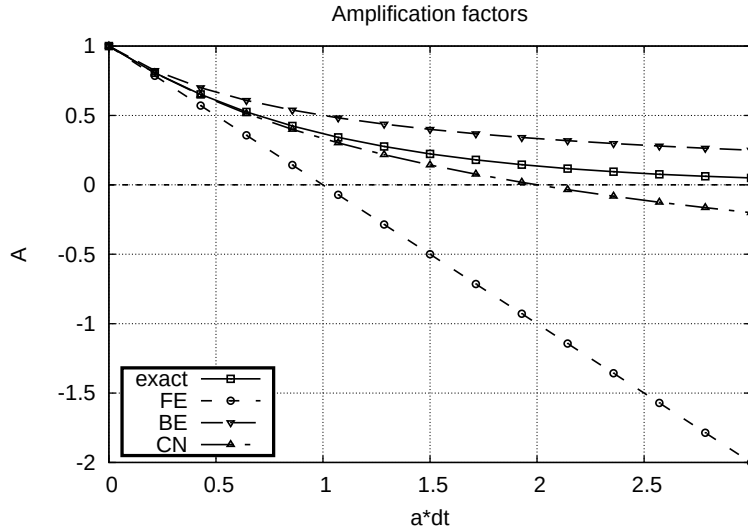


Figure 14: Comparison of amplification factors.

.5 Series expansion of amplification factors

s an alternative to the visual understanding inherent in Figure 14, there is a rong tradition in numerical analysis to establish formulas for the approximation rors when the discretization parameter, here Δt, becomes small. In the present ase we let p be our small discretization parameter, and it makes sense to simplify ie expressions for A and A_e by using Taylor polynomials around $p = 0$. The aylor polynomials are accurate for small p and greatly simplifies the comparison f the analytical expressions since we then can compare polynomials, term by erm.

Calculating the Taylor series for A_e is easily done by hand, but the three ersions of A for $\theta = 0, 1, \frac{1}{2}$ lead to more cumbersome calculations. Nowadays, nalytical computations can benefit greatly by symbolic computer algebra soft- are. The Python package `sympy` represents a powerful computer algebra system, ot yet as sophisticated as the famous Maple and Mathematica systems, but ee and very easy to integrate with our numerical computations in Python.

When using `sympy`, it is convenient to enter the interactive Python mode here we can write expressions and statements and immediately see the results. ere is a simple example. We strongly recommend to use `isympy` (or `ipython`) r such interactive sessions.

Let us illustrate `sympy` with a standard Python shell syntax (`>>>` prompt) to mpute a Taylor polynomial approximation to e^{-p}:

```
>>> from sympy import *
>>> # Create p as a mathematical symbol with name 'p'
>>> p = Symbol('p')
>>> # Create a mathematical expression with p
>>> A_e = exp(-p)
>>>
>>> # Find the first 6 terms of the Taylor series of A_e
>>> A_e.series(p, 0, 6)
1 + (1/2)*p**2 - p - 1/6*p**3 - 1/120*p**5 + (1/24)*p**4 + O(p**6)
```

ines with `>>>` represent input lines and lines without this prompt represents ie result of computations (note that `isympy` and `ipython` apply other prompts, ut in this text we always apply `>>>` for interactive Python computing). Apart om the order of the powers, the computed formula is easily recognized as the eginning of the Taylor series for e^{-p}.

Let us define the numerical amplification factor where p and θ enter the rmula as symbols:

```
>>> theta = Symbol('theta')
>>> A = (1-(1-theta)*p)/(1+theta*p)
```

o work with the factor for the Backward Euler scheme we can substitute the alue 1 for `theta`:


```
>>> A.subs(theta, 1)
1/(1 + p)
```

Similarly, we can replace `theta` by 1/2 for Crank-Nicolson, preferably u exact rational representation of 1/2 in `sympy`:

```
>>> half = Rational(1,2)
>>> A.subs(theta, half)
1/(1 + (1/2)*p)*(1 - 1/2*p)
```

The Taylor series of the amplification factor for the Crank-Nicolson can be computed as

```
>>> A.subs(theta, half).series(p, 0, 4)
1 + (1/2)*p**2 - p - 1/4*p**3 + O(p**4)
```

We are now in a position to compare Taylor series:

```
>>> FE = A_e.series(p, 0, 4) - A.subs(theta, 0).series(p, 0, 4)
>>> BE = A_e.series(p, 0, 4) - A.subs(theta, 1).series(p, 0, 4)
>>> CN = A_e.series(p, 0, 4) - A.subs(theta, half).series(p, 0, 4
>>> FE
(1/2)*p**2 - 1/6*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (5/6)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

From these expressions we see that the error $A - A_e \sim \mathcal{O}(p^2)$ for the I and Backward Euler schemes, while $A - A_e \sim \mathcal{O}(p^3)$ for the Crank-N scheme. It is the *leading order term*, i.e., the term of the lowest order (pol degree), that is of interest, because as $p \to 0$, this term is (much) bigg the higher-order terms (think of $p = 0.01$: p is a hundred times larger t

Now, a is a given parameter in the problem, while Δt is what we c One therefore usually writes the error expressions in terms Δt. When th

$$A - A_e = \begin{cases} \mathcal{O}(\Delta t^2), & \text{Forward and Backward Euler,} \\ \mathcal{O}(\Delta t^3), & \text{Crank-Nicolson} \end{cases}$$

We say that the Crank-Nicolson scheme has an error in the ampli factor of order Δt^3, while the two other schemes are of order Δt^2 in tl quantity. What is the significance of the order expression? If we ha the error in amplification factor at a time level will be reduced by a f 4 in the Forward and Backward Euler schemes, and by a factor of 8 Crank-Nicolson scheme. That is, as we reduce Δt to obtain more a results, the Crank-Nicolson scheme reduces the error more efficiently t other schemes.

.6 The fraction of numerical and exact amplification factors

n alternative comparison of the schemes is to look at the ratio A/A_e, or the ror $1 - A/A_\mathrm{e}$ in this ratio:

```
>>> FE = 1 - (A.subs(theta, 0)/A_e).series(p, 0, 4)
>>> BE = 1 - (A.subs(theta, 1)/A_e).series(p, 0, 4)
>>> CN = 1 - (A.subs(theta, half)/A_e).series(p, 0, 4)
>>> FE
(1/2)*p**2 + (1/3)*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (1/3)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

he leading-order terms have the same powers as in the analysis of $A - A_\mathrm{e}$.

.7 The global error at a point

he error in the amplification factor reflects the error when progressing from me level t_n to t_{n-1}. To investigate the real error at a point, known as the *obal error*, we look at $e^n = u^n - u_\mathrm{e}(t_n)$ for some n and Taylor expand the athematical expressions as functions of $p = a\Delta t$:

```
>>> n = Symbol('n')
>>> u_e = exp(-p*n)
>>> u_n = A**n
>>> FE = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
>>> BE = u_e.series(p, 0, 4) - u_n.subs(theta, 1).series(p, 0, 4)
>>> CN = u_e.series(p, 0, 4) - u_n.subs(theta, half).series(p, 0, 4)
>>> FE
(1/2)*n*p**2 - 1/2*n**2*p**3 + (1/3)*n*p**3 + O(p**4)
>>> BE
(1/2)*n**2*p**3 - 1/2*n*p**2 + (1/3)*n*p**3 + O(p**4)
>>> CN
(1/12)*n*p**3 + O(p**4)
```

or a fixed time t, the parameter n in these expressions increases as $p \to 0$ since $= n\Delta t =$ const and hence n must increase like Δt^{-1}. With n substituted y $t/\Delta t$ in the leading-order error terms, these become $\frac{1}{2}na^2\Delta t^2 = \frac{1}{2}ta^2\Delta t$ for ıe Forward and Backward Euler scheme, and $\frac{1}{12}na^3\Delta t^3 = \frac{1}{12}ta^3\Delta t^2$ for the rank-Nicolson scheme. The global error is therefore of second order (in Δt) for ıe latter scheme and of first order for the former schemes.

When the global error $e^n \to 0$ as $\Delta t \to 0$, we say that the scheme is *convergent*. means that the numerical solution approaches the exact solution as the mesh refined, and this is a much desired property of a numerical method.

.8 Integrated errors

is common to study the norm of the numerical error, as explained in detail in ection 2.4. The L^2 norm can be computed by treating e^n as a function of t in

sympy and performing symbolic integration. For the Forward Euler sch have

```
p, n, a, dt, t, T, theta = symbols('p n a dt t T 'theta')
A = (1-(1-theta)*p)/(1+theta*p)
u_e = exp(-p*n)
u_n = A**n
error = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
# Introduce t and dt instead of n and p
error = error.subs('n', 't/dt').subs(p, 'a*dt')
error = error.as_leading_term(dt) # study only the first term
print error
error_L2 = sqrt(integrate(error**2, (t, 0, T)))
print error_L2
```

The output reads

```
sqrt(30)*sqrt(T**3*a**4*dt**2*(6*T**2*a**2 - 15*T*a + 10))/60
```

which means that the L^2 error behaves like $a^2\Delta t$.

Strictly speaking, the numerical error is only defined at the mesh poi makes most sense to compute the ℓ^2 error

$$||e^n||_{\ell^2} = \sqrt{\Delta t \sum_{n=0}^{N_t} (u_\mathrm{e}(t_n) - u^n)^2}\,.$$

We have obtained an exact analytical expressions for the error at $t =$ here we use the leading-order error term only since we are mostly inter how the error behaves as a polynomial in Δt, and then the leading ord will dominate. For the Forward Euler scheme, $u_\mathrm{e}(t_n) - u^n \approx \frac{1}{2}np^2$, and

$$||e^n||_{\ell^2}^2 = \Delta t \sum_{n=0}^{N_t} \frac{1}{4}n^2p^4 = \Delta t \frac{1}{4}p^4 \sum_{n=0}^{N_t} n^2\,.$$

Now, $\sum_{n=0}^{N_t} n^2 \approx \frac{1}{3}N_t^3$. Using this approximation, setting $N_t = T/$ taking the square root gives the expression

$$||e^n||_{\ell^2} = \frac{1}{2}\sqrt{\frac{T^3}{3}}a^2\Delta t\,.$$

Calculations for the Backward Euler scheme are very similar and prov same result, while the Crank-Nicolson scheme leads to

$$||e^n||_{\ell^2} = \frac{1}{12}\sqrt{\frac{T^3}{3}}a^3\Delta t^2\,.$$

Summary of errors.

Both the point-wise and the time-integrated true errors are of second order in Δt for the Crank-Nicolson scheme and of first order in Δt for the Forward Euler and Backward Euler schemes.

.9 Truncation error

he truncation error is a very frequently used error measure for finite difference ıethods. It is defined as *the error in the difference equation that arises when ıserting the exact solution*. Contrary to many other error measures, e.g., the ue error $e^n = u_{\rm e}(t_n) - u^n$, the truncation error is a quantity that is easily ɔmputable.

Let us illustrate the calculation of the truncation error for the Forward Euler cheme. We start with the difference equation on operator form,

$$[D_t u = -au]^n,$$

e.,

$$\frac{u^{n+1} - u^n}{\Delta t} = -au^n\,.$$

he idea is to see how well the exact solution $u_{\rm e}(t)$ fulfills this equation. Since $_{\rm e}(t)$ in general will not obey the discrete equation, error in the discrete equation, alled a *residual*, denoted here by R^n:

$$R^n = \frac{u_{\rm e}(t_{n+1}) - u_{\rm e}(t_n)}{\Delta t} + au_{\rm e}(t_n)\,. \tag{55}$$

he residual is defined at each mesh point and is therefore a mesh function with superscript n.

The interesting feature of R^n is to see how it depends on the discretization arameter Δt. The tool for reaching this goal is to Taylor expand $u_{\rm e}$ around the oint where the difference equation is supposed to hold, here $t = t_n$. We have ıat

$$u_{\rm e}(t_{n+1}) = u_{\rm e}(t_n) + u'_{\rm e}(t_n)\Delta t + \frac{1}{2}u''_{\rm e}(t_n)\Delta t^2 + \cdots$$

ıserting this Taylor series in (55) gives

$$R^n = u'_{\rm e}(t_n) + \frac{1}{2}u''_{\rm e}(t_n)\Delta t + \ldots + au_{\rm e}(t_n)\,.$$

ow, $u_{\rm e}$ fulfills the ODE $u'_{\rm e} = -au_{\rm e}$ such that the first and last term cancels nd we have

$$R^n \approx \frac{1}{2}u''_{\rm e}(t_n)\Delta t\,.$$

This R^n is the *truncation error*, which for the Forward Euler is seen first order in Δt.

The above procedure can be repeated for the Backward Euler and the Nicolson schemes. We start with the scheme in operator notation, write detail, Taylor expand $u_{\rm e}$ around the point $\tilde{t}$ at which the difference equ defined, collect terms that correspond to the ODE (here $u'_{\rm e} + au_{\rm e}$), and the remaining terms as the residual R, which is the truncation erro Backward Euler scheme leads to

$$R^n \approx -\frac{1}{2}u''_{\rm e}(t_n)\Delta t,$$

while the Crank-Nicolson scheme gives

$$R^{n+\frac{1}{2}} \approx \frac{1}{24}u'''_{\rm e}(t_{n+\frac{1}{2}})\Delta t^2\,.$$

The *order* r of a finite difference scheme is often defined through the term Δt^r in the truncation error. The above expressions point out t Forward and Backward Euler schemes are of first order, while Crank-N is of second order. We have looked at other error measures in other s like the error in amplification factor and the error $e^n = u_{\rm e}(t_n) -$ expressed these error measures in terms of Δt to see the order of the Normally, calculating the truncation error is more straightforward than the expressions for other error measures and therefore the easiest way to e the order of a scheme.

3.10 Consistency, stability, and convergence

Three fundamental concepts when solving differential equations by nu methods are consistency, stability, and convergence. We shall briefly tou concepts below in the context of the present model problem.

Consistency means that the error in the difference equation, measured the truncation error, goes to zero as $\Delta t \to 0$. Since the truncatio tells how well the exact solution fulfills the difference equation, and th solution fulfills the differential equation, consistency ensures that the di equation approaches the differential equation in the limit. The expression truncation errors in the previous section are all proportional to Δt or Δt they vanish as $\Delta t \to 0$, and all the schemes are consistent. Lack of con implies that we actually solve a different differential equation in the limit than we aim at.

Stability means that the numerical solution exhibits the same qua properties as the exact solution. This is obviously a feature we want the nu solution to have. In the present exponential decay model, the exact sol monotone and decaying. An increasing numerical solution is not in acc with the decaying nature of the exact solution and hence unstable. We say that an oscillating numerical solution lacks the property of mono of the exact solution and is also unstable. We have seen that the Ba

uler scheme always leads to monotone and decaying solutions, regardless of Δt, nd is hence stable. The Forward Euler scheme can lead to increasing solutions nd oscillating solutions if Δt is too large and is therefore unstable unless Δt is ufficiently small. The Crank-Nicolson can never lead to increasing solutions and as no problem to fulfill that stability property, but it can produce oscillating olutions and is unstable in that sense, unless Δt is sufficiently small.

Convergence implies that the global (true) error mesh function $e^n = u_{\mathrm{e}}(t_n) - {}^n \to 0$ as $\Delta t \to 0$. This is really what we want: the numerical solution gets as ose to the exact solution as we request by having a sufficiently fine mesh.

Convergence is hard to establish theoretically, except in quite simple problems ke the present one. Stability and consistency are much easier to calculate. A ajor breakthrough in the understanding of numerical methods for differential quations came in 1956 when Lax and Richtmeyer established equivalence etween convergence on one hand and consistency and stability on the other (the ax equivalence theorem[19]). In practice it meant that one can first establish that method is stable and consistent, and then it is automatically convergent (which much harder to establish). The result holds for linear problems only, and in ne world of nonlinear differential equations the relations between consistency, tability, and convergence are much more complicated.

We have seen in the previous analysis that the Forward Euler, Backward uler, and Crank-Nicolson schemes are convergent ($e^n \to 0$), that they are onsistent ($R^n \to 0$, and that they are stable under certain conditions on the ze of Δt. We have also derived explicit mathematical expressions for e^n, the runcation error, and the stability criteria.

Exercises

Exercise 1: Visualize the accuracy of finite differences

he purpose of this exercise is to visualize the accuracy of finite difference pproximations of the derivative of a given function. For any finite difference pproximation, take the Forward Euler difference as an example, and any specific unction, take $u = e^{-at}$, we may introduce an error fraction

$$E = \frac{[D_t^+ u]^n}{u'(t_n)} = \frac{\exp\left(-a(t_n + \Delta t)\right) - \exp\left(-at_n\right)}{-a\exp\left(-at_n\right)} = -\frac{1}{a\Delta t}\left(\exp\left(-a\Delta t\right) - 1\right),$$

nd view E as a function of Δt. We expect that $\lim_{\Delta t\to 0} E = 1$, while E may eviate significantly from unity for large Δt. How the error depends on Δt is est visualized in a graph where we use a logarithmic scale on for Δt, so we an cover many orders of magnitude of that quantity. Here is a code segment reating an array of 100 intervals, on the logarithmic scale, ranging from 10^{-6} o 1 and then plotting E versus $p = a\Delta t$ with logarithmic scale on the Δt axis:

[19] http://en.wikipedia.org/wiki/Lax_equivalence_theorem


```
from numpy import logspace, exp
from matplotlib.pyplot import plot
p = logspace(-6, 1, 101)
y = -(exp(-p)-1)/p
semilog(p, y)
```

Illustrate such errors for the finite difference operators $[D_t^+ u]^n$ (forward), (backward), and $[D_t u]^n$ (centered).

Perform a Taylor series expansions of the error fractions and find the order r in the expressions of type $1 + C\Delta t^r + \mathcal{O}(\Delta t^{r+1})$, where C is some c Filename: `decay_plot_fd_error.py`.

Exercise 2: Explore the θ-rule for exponential growth

This exercise asks you to solve the ODE $u' = -au$ with $a < 0$ such t ODE models exponential growth instead of exponential decay. A centra is to investigate numerical artifacts and non-physical solution behavior.

a) Run experiments with $\theta = 0, 0.5, 1$ for various values of Δt to numerical artifacts. Recall that the exact solution is a monotone, function when $a < 0$. Oscillations or significantly wrong growth are wrong qualitative behavior, which can be used to define a stability crit

Use the insight to select a few values of Δt that demonstrate all t numerical artifacts for the three different schemes ($\theta = 0, 0.5, 1$). Keep a these experiments. Filename: `growth_demo.py`.

b) Write up the amplification factor and plot it for $\theta = 0, 0.5, 1$ togeth the exact one for $a\Delta t < 0$. Use the plot to explain the observations mad experiments.

Hint. Modify the `decay_ampf_plot.py`[20] code.
Filename: `growth_ampf.py`.

5 Model extensions

It is time to consider generalizations of the simple decay model $u =$ - also to look at additional numerical solution methods.

5.1 Generalization: including a variable coefficient

In the ODE for decay, $u' = -au$, we now consider the case where a dep time:

$$u'(t) = -a(t)u(t), \quad t \in (0, T], \quad u(0) = I\,.$$

[20] http://tinyurl.com/jvzzcfn/decay/decay_ampf_plot.py

A Forward Euler scheme consist of evaluating (56) at $t = t_n$ and approximating the derivative with a forward difference $[D_t^+ u]^n$:

$$\frac{u^{n+1} - u^n}{\Delta t} = -a(t_n)u^n \,. \tag{57}$$

The Backward Euler scheme becomes

$$\frac{u^n - u^{n-1}}{\Delta t} = -a(t_n)u^n \,. \tag{58}$$

The Crank-Nicolson method builds on sampling the ODE at $t_{n+\frac{1}{2}}$. We can evaluate a at $t_{n+\frac{1}{2}}$ and use an average for u at times t_n and t_{n+1}:

$$\frac{u^{n+1} - u^n}{\Delta t} = -a(t_{n+\frac{1}{2}})\frac{1}{2}(u^n + u^{n+1}) \,. \tag{59}$$

Alternatively, we can use an average for the product au:

$$\frac{u^{n+1} - u^n}{\Delta t} = -\frac{1}{2}(a(t_n)u^n + a(t_{n+1})u^{n+1}) \,. \tag{60}$$

The θ-rule unifies the three mentioned schemes. One version is to have a evaluated at $t_{n+\theta}$,

$$\frac{u^{n+1} - u^n}{\Delta t} = -a((1-\theta)t_n + \theta t_{n+1})((1-\theta)u^n + \theta u^{n+1}) \,. \tag{61}$$

Another possibility is to apply a weighted average for the product au,

$$\frac{u^{n+1} - u^n}{\Delta t} = -(1-\theta)a(t_n)u^n - \theta a(t_{n+1})u^{n+1} \,. \tag{62}$$

With the finite difference operator notation the Forward Euler and Backward Euler schemes can be summarized as

$$[D_t^+ u = -au]^n, \tag{63}$$
$$[D_t^- u = -au]^n \,. \tag{64}$$

The Crank-Nicolson and θ schemes depend on whether we evaluate a at the simple point for the ODE or if we use an average. The various versions are written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}}, \tag{65}$$
$$[D_t u = -\overline{au}^t]^{n+\frac{1}{2}}, \tag{66}$$
$$[D_t u = -a\overline{u}^{t,\theta}]^{n+\theta}, \tag{67}$$
$$[D_t u = -\overline{au}^{t,\theta}]^{n+\theta} \,. \tag{68}$$

5.2 Generalization: including a source term

A further extension of the model ODE is to include a source term $b(t)$:

$$u'(t) = -a(t)u(t) + b(t), \quad t \in (0,T], \quad u(0) = I \,.$$

Schemes. The time point where we sample the ODE determines where a and b are evaluated. For the Crank-Nicolson scheme and the θ-rule we have a choice of whether to evaluate $a(t)$ and $b(t)$ at the correct point or use an average. The chosen strategy becomes particularly clear if we write up the schemes in short operator notation:

$$[D_t^+ u = -au + b]^n,$$
$$[D_t^- u = -au + b]^n,$$
$$[D_t u = -a\overline{u}^t + b]^{n+\frac{1}{2}},$$
$$[D_t u = \overline{-au + b}^t]^{n+\frac{1}{2}},$$
$$[D_t u = -a\overline{u}^{t,\theta} + b]^{n+\theta},$$
$$[D_t u = \overline{-au + b}^{t,\theta}]^{n+\theta} \,.$$

5.3 Implementation of the generalized model problem

Deriving the θ-rule formula. Writing out the θ-rule in (75), using (33), we get

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta(-a^{n+1}u^{n+1} + b^{n+1})) + (1-\theta)(-a^n u^n + b^n)),$$

where a^n means evaluating a at $t = t_n$ and similar for a^{n+1}, b^n, and b^{n+1}. We solve for u^{n+1}:

$$u^{n+1} = ((1 - \Delta t(1-\theta)a^n)u^n + \Delta t(\theta b^{n+1} + (1-\theta)b^n))(1 + \Delta t\theta a^{n+1})^{-}$$

The Python code. Here is a suitable implementation of (76) where $a(t)$ and $b(t)$ are given as Python functions:

```
def solver(I, a, b, T, dt, theta):
    """
    Solve u'=-a(t)*u + b(t), u(0)=I,
    for t in (0,T] with steps of dt.
    a and b are Python functions of t.
    """
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh
```

```
    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = ((1 - dt*(1-theta)*a(t[n]))*u[n] + \
                  dt*(theta*b(t[n+1]) + (1-theta)*b(t[n])))/\
                  (1 + dt*theta*a(t[n+1]))
    return u, t
```

his function is found in the file `decay_vc.py`[21] (`vc` stands for "variable coeffi-ents").

oding of variable coefficients. The `solver` function shown above demands ne arguments `a` and `b` to be Python functions of time `t`, say

```
def a(t):
    return a_0 if t < tp else k*a_0

def b(t):
    return 1
```

ere, `a(t)` has three parameters `a0`, `tp`, and `k`, which must be global variables. better implementation is to represent `a` by a class where the parameters are ttributes and a *special method* `__call__` evaluates $a(t)$:

```
class A:
    def __init__(self, a0=1, k=2):
        self.a0, self.k = a0, k

    def __call__(self, t):
        return self.a0 if t < self.tp else self.k*self.a0

a = A(a0=2, k=1)  # a behaves as a function a(t)
```

For quick tests it is cumbersome to write a complete function or a class. The *mbda function* construction in Python is then convenient. For example,

```
a = lambda t: a_0 if t < tp else k*a_0
```

equivalent to the `def a(t):` definition above. In general,

```
f = lambda arg1, arg2, ...: expressin
```

equivalent to

```
def f(arg1, arg2, ...):
    return expression
```

ne can use lambda functions directly in calls. Say we want to solve $u' = -u+1$, $(0) = 2$:

[21] `http://tinyurl.com/jvzzcfn/decay/decay_vc.py`


```
u, t = solver(2, lambda t: 1, lambda t: 1, T, dt, theta)
```

A lambda function can appear anywhere where a variable can appear.

5.4 Verifying a constant solution

A very useful partial verification method is to construct a test proble a very simple solution, usually $u = \text{const}$. Especially the initial debug a program code can benefit greatly from such tests, because 1) all numerical methods will exactly reproduce a constant solution, 2) many intermediate calculations are easy to control for a constant u, and 3) constant u can uncover many bugs in an implementation.

The only constant solution for the problem $u' = -au$ is $u = 0$, but to bugs can escape from that trivial solution. It is much better to searc problem where $u = C = \text{const} \neq 0$. Then $u' = -a(t)u + b(t)$ is more appr with $u = C$ we can choose any $a(t)$ and set $b = a(t)C$ and $I = C$. An app nose test is

```
import nose.tools as nt

def test_constant_solution():
    """
    Test problem where u=u_const is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return u_const

    def a(t):
        return 2.5*(1+t**3)  # can be arbitrary

    def b(t):
        return a(t)*u_const

    u_const = 2.15
    theta = 0.4; I = u_const; dt = 4
    Nt = 4  # enough with a few steps
    u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
    print u
    u_e = exact_solution(t)
    difference = abs(u_e - u).max()  # max deviation
    nt.assert_almost_equal(difference, 0, places=14)
```

An interesting question is what type of bugs that will make the comp deviate from the exact solution C. Fortunately, the updating formula initial condition must be absolutely correct for the test to pass! Any att make a wrong indexing in terms like `a(t[n])` or any attempt to intro erroneous factor in the formula creates a solution that is different from

5.5 Verification via manufactured solutions

Following the idea of the previous section, we can choose any formula exact solution, insert the formula in the ODE problem and fit the data a

nd I to make the chosen formula fulfill the equation. This powerful technique or generating exact solutions is very useful for verification purposes and known s the *method of manufactured solutions*, often abbreviated MMS.

One common choice of solution is a linear function in the independent ariable(s). The rationale behind such a simple variation is that almost any elevant numerical solution method for differential equation problems is able to eproduce the linear function exactly to machine precision (if u is about unity n size; precision is lost if u take on large values, see Exercise 3). The linear olution also makes some stronger demands to the numerical method and the nplementation than the constant solution used in Section 5.4, at least in more omplicated applications. However, the constant solution is often ideal for initial ebugging before proceeding with a linear solution.

We choose a linear solution $u(t) = ct + d$. From the initial condition it follows hat $d = I$. Inserting this u in the ODE results in

$$c = -a(t)u + b(t)\,.$$

ny function $u = ct + I$ is then a correct solution if we choose

$$b(t) = c + a(t)(ct + I)\,.$$

/ith this $b(t)$ there are no restrictions on $a(t)$ and c.

Let prove that such a linear solution obeys the numerical schemes. To this nd, we must check that $u^n = ca(t_n)(ct_n + I)$ fulfills the discrete equations. For hese calculations, and later calculations involving linear solutions inserted in nite difference schemes, it is convenient to compute the action of a difference perator on a linear function t:

$$[D_t^+ t]^n = \frac{t_{n+1} - t_n}{\Delta t} = 1, \tag{78}$$

$$[D_t^- t]^n = \frac{t_n - t_{n-1}}{\Delta t} = 1, \tag{79}$$

$$[D_t t]^n = \frac{t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}}}{\Delta t} = \frac{(n+\frac{1}{2})\Delta t - (n-\frac{1}{2})\Delta t}{\Delta t} = 1\,. \tag{80}$$

learly, all three finite difference approximations to the derivative are exact for $(t) = t$ or its mesh function counterpart $u^n = t_n$.

The difference equation for the Forward Euler scheme

$$[D_t^+ u = -au + b]^n,$$

ith $a^n = a(t_n)$, $b^n = c + a(t_n)(ct_n + I)$, and $u^n = ct_n + I$ then results in

$$c = -a(t_n)(ct_n + I) + c + a(t_n)(ct_n + I) = c$$

hich is always fulfilled. Similar calculations can be done for the Backward uler and Crank-Nicolson schemes, or the θ-rule for that matter. In all cases,

$u^n = ct_n + I$ is an exact solution of the discrete equations. That is should expect that $u^n - u_e(t_n) = 0$ mathematically and $|u^n - u_e(t_n)|$ l a small number about the machine precision for $n = 0, \ldots, N_t$.

The following function offers an implementation of this verification te on a linear exact solution:

```
def test_linear_solution():
    """
    Test problem where u=c*t+I is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return c*t + I

    def a(t):
        return t**0.5  # can be arbitrary

    def b(t):
        return c + a(t)*exact_solution(t)

    theta = 0.4; I = 0.1; dt = 0.1; c = -0.5
    T = 4
    Nt = int(T/dt)  # no of steps
    u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
    u_e = exact_solution(t)
    difference = abs(u_e - u).max()  # max deviation
    print difference
    # No of decimal places for comparison depend on size of c
    nt.assert_almost_equal(difference, 0, places=14)
```

Any error in the updating formula makes this test fail!

Choosing more complicated formulas as the exact solution, say $\cos(t)$, make the numerical and exact solution coincide to machine precision, finite differencing of $\cos(t)$ does not exactly yield the exact derivative $-\sin$ such cases, the verification procedure must be based on measuring the conv rates as exemplified in Section **??**. Convergence rates can be computed as one has an exact solution of a problem that the solver can be tested this can always be obtained by the method of manufactured solutions.

5.6 Extension to systems of ODEs

Many ODE models involves more than one unknown function and mc one equation. Here is an example of two unknown functions $u(t)$ and v

$$u' = au + bv,$$
$$v' = cu + dv,$$

for constants a, b, c, d. Applying the Forward Euler method to each e results in simple updating formula

$$u^{n+1} = u^n + \Delta t(au^n + bv^n), \tag{83}$$
$$v^{n+1} = u^n + \Delta t(cu^n + dv^n)\,. \tag{84}$$

)n the other hand, the Crank-Nicolson or Backward Euler schemes result in a
× 2 linear system for the new unknowns. The latter schemes gives

$$u^{n+1} = u^n + \Delta t(au^{n+1} + bv^{n+1}), \tag{85}$$
$$v^{n+1} = v^n + \Delta t(cu^{n+1} + dv^{n+1})\,. \tag{86}$$

'ollecting u^{n+1} as well as v^{n+1} on the left-hand side results in

$$(1 - \Delta ta)u^{n+1} + bv^{n+1} = u^n, \tag{87}$$
$$cu^{n+1} + (1 - \Delta td)v^{n+1} = v^n, \tag{88}$$

hich is a system of two coupled, linear, algebraic equations in two unknowns.

5 General first-order ODEs

Ve now turn the attention to general, nonlinear ODEs and systems of such)DEs. Our focus is on numerical methods that can be readily reused for time-iscretization PDEs, and diffusion PDEs in particular. The methods are just riefly listed, and we refer to the rich literature for more detailed descriptions nd analysis - the books [6, 1, 2, 3] are all excellent resources on numerical ıethods for ODEs. We also demonstrate the Odespy Python interface to a ınge of different software for general first-order ODE systems.

.1 Generic form of first-order ODEs

)DEs are commonly written in the generic form

$$u' = f(u,t), \quad u(0) = I, \tag{89}$$

here $f(u,t)$ is some prescribed function. As an example, our most general kponential decay model (69) has $f(u,t) = -a(t)u(t) + b(t)$.

The unknown u in (89) may either be a scalar function of time t, or a vector ılued function of t in case of a *system of ODEs* with m unknown components:

$$u(t) = (u^{(0)}(t), u^{(1)}(t), \ldots, u^{(m-1)}(t))\,.$$

ı that case, the right-hand side is vector-valued function with m components,

$$\begin{aligned} f(u,t) = (&f^{(0)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)),\\ &f^{(1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)),\\ &\vdots,\\ &f^{(m-1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)))\,. \end{aligned}$$

Actually, any system of ODEs can be written in the form (89), but order ODEs then need auxiliary unknown functions to enable convers first-order system.

Next we list some well-known methods for $u' = f(u,t)$, valid both for ODE (scalar u) and systems of ODEs (vector u). The choice of met inspired by the kind of schemes that are popular also for time discretiz partial differential equations.

6.2 The θ-rule

The θ-rule scheme applied to $u' = f(u,t)$ becomes

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta f(u^{n+1}, t_{n+1}) + (1-\theta)f(u^n, t_n)\,.$$

Bringing the unknown u^{n+1} to the left-hand side and the known terms right-hand side gives

$$u^{n+1} - \Delta t\theta f(u^{n+1}, t_{n+1}) = u^n + \Delta t(1-\theta)f(u^n, t_n)\,.$$

For a general f (not linear in u), this equation is *nonlinear* in the unknov unless $\theta = 0$. For a scalar ODE ($m = 1$), we have to solve a single nc algebraic equation for u^{n+1}, while for a system of ODEs, we get a sy coupled, nonlinear algebraic equations. Newton's method is a popular approach in both cases. Note that with the Forward Euler scheme (θ do not have to deal with nonlinear equations, because in that case we explicit updating formula for u^{n+1}. This is known as an *explicit* schem $\theta \neq 1$ we have to solve (systems of) algebraic equations, and the scheme to be *implicit*.

6.3 An implicit 2-step backward scheme

The implicit backward method with 2 steps applies a three-level ba difference as approximation to $u'(t)$,

$$u'(t_{n+1}) \approx \frac{3u^{n+1} - 4u^n + u^{n-1}}{2\Delta t},$$

which is an approximation of order Δt^2 to the first derivative. The r scheme for $u' = f(u,t)$ reads

$$u^{n+1} = \frac{4}{3}u^n - \frac{1}{3}u^{n-1} + \frac{2}{3}\Delta t f(u^{n+1}, t_{n+1})\,.$$

Higher-order versions of the scheme (92) can be constructed by includi time levels. These schemes are known as the Backward Differentiation F (BDF), and the particular version (92) is often referred to as BDF2.

Note that the scheme (92) is implicit and requires solution of nc equations when f is nonlinear in u. The standard 1st-order Backwar method or the Crank-Nicolson scheme can be used for the first step.

.4 Leapfrog schemes

'he ordinary Leapfrog scheme. The derivative of u at some point t_n can e approximated by a central difference over two time steps,

$$u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n \tag{93}$$

hich is an approximation of second order in Δt. The scheme can then be ritten as

$$[D_{2t}u = f(u,t)]^n,$$

ı operator notation. Solving for u^{n+1} gives

$$u^{n+1} = u^{n-1} + \Delta t f(u^n, t_n)\,. \tag{94}$$

)bserve that (94) is an explicit scheme, and that a nonlinear f (in u) is trivial ɔ handle since it only involves the known u^n value. Some other scheme must e used as starter to compute u^1, preferably the Forward Euler scheme since it also explicit.

'he filtered Leapfrog scheme. Unfortunately, the Leapfrog scheme (94) ill develop growing oscillations with time (see Problem 8)[[[. A remedy for ıch undesired oscillations is to introduce a *filtering technique*. First, a standard eapfrog step is taken, according to (94), and then the previous u^n value is djusted according to

$$u^n \;\leftarrow\; u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})\,. \tag{95}$$

he γ-terms will effectively damp oscillations in the solution, especially those ith short wavelength (like point-to-point oscillations). A common choice of γ is .6 (a value used in the famous NCAR Climate Model).

.5 The 2nd-order Runge-Kutta method

he two-step scheme

$$u^* = u^n + \Delta t f(u^n, t_n), \tag{96}$$

$$u^{n+1} = u^n + \Delta t \frac{1}{2}\left(f(u^n, t_n) + f(u^*, t_{n+1})\right), \tag{97}$$

ssentially applies a Crank-Nicolson method (97) to the ODE, but replaces the erm $f(u^{n+1}, t_{n+1})$ by a prediction $f(u^*, t_{n+1})$ based on a Forward Euler step)6). The scheme (96)-(97) is known as Huen's method, but is also a 2nd-order unge-Kutta method. The scheme is explicit, and the error is expected to behave s Δt^2.

6.6 A 2nd-order Taylor-series method

One way to compute u^{n+1} given u^n is to use a Taylor polynomial. We m up a polynomial of 2nd degree:

$$u^{n+1} = u^n + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2\,.$$

From the equation $u' = f(u,t)$ it follows that the derivatives of u can be e in terms of f and its derivatives:

$$\begin{aligned} u'(t_n) &= f(u^n, t_n), \\ u''(t_n) &= \frac{\partial f}{\partial u}(u^n, t_n)u'(t_n) + \frac{\partial f}{\partial t} \\ &= f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}, \end{aligned}$$

resulting in the scheme

$$u^{n+1} = u^n + f(u^n, t_n)\Delta t + \frac{1}{2}\left(f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}\right)\Delta t^2\,.$$

More terms in the series could be included in the Taylor polynomial t methods of higher order than 2.

6.7 The 2nd- and 3rd-order Adams-Bashforth sche

The following method is known as the 2nd-order Adams-Bashforth sch

$$u^{n+1} = u^n + \frac{1}{2}\Delta t\left(3f(u^n, t_n) - f(u^{n-1}, t_{n-1})\right)\,.$$

The scheme is explicit and requires another one-step scheme to compute Forward Euler scheme or Heun's method, for instance). As the name the scheme is of order Δt^2.

Another explicit scheme, involving four time levels, is the 3rd-order Bashforth scheme

$$u^{n+1} = u^n + \frac{1}{12}\left(23f(u^n, t_n) - 16f(u^{n-1}, t_{n-1}) + 5f(u^{n-2}, t_{n-2})\right)\,.$$

The numerical error is of order Δt^3, and the scheme needs some met computing u^1 and u^2.

More general, higher-order Adams-Bashforth schemes (also called *Adams methods*) compute u^{n+1} as a linear combination of f at k previc steps:

$$u^{n+1} = u^n + \sum_{j=0}^{k}\beta_j f(u^{n-j}, t_{n-j}),$$

where β_j are known coefficients.

.8 The 4th-order Runge-Kutta method

he perhaps most widely used method to solve ODEs is the 4th-order Runge-
utta method, often called RK4. Its derivation is a nice illustration of common
umerical approximation strategies, so let us go through the steps in detail.

The starting point is to integrate the ODE $u' = f(u,t)$ from t_n to t_{n+1}:

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(u(t),t)dt\,.$$

Ve want to compute $u(t_{n+1})$ and regard $u(t_n)$ as known. The task is to find
ood approximations for the integral, since the integrand involves the unknown
between t_n and t_{n+1}.

The integral can be approximated by the famous Simpson's rule[22]:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 4f^{n+\frac{1}{2}} + f^{n+1}\right).$$

he problem now is that we do not know $f^{n+\frac{1}{2}} = f(u^{n+\frac{1}{2}}, t_{n+1/2})$ and $f^{n+1} = f(u^{n+1}, t_{n+1})$ as we know only u^n and hence f^n. The idea is to use various approx-
nations for $f^{n+\frac{1}{2}}$ and f^{n+1} based on using well-known schemes for the ODE in
ne intervals $[t_n, t_{n+1/2}]$ and $[t_n, t_{n+1}]$. We split the integral approximation into
our terms:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right),$$

here $\hat{f}^{n+\frac{1}{2}}$, $\tilde{f}^{n+\frac{1}{2}}$, and $\bar{f}^{n+1}$ are approximations to $f^{n+\frac{1}{2}}$ and f^{n+1} that can be
ased on already computed quantities. For $\hat{f}^{n+\frac{1}{2}}$ we can apply an approximation
o $u^{n+\frac{1}{2}}$ using the Forward Euler method with step $\frac{1}{2}\Delta t$:

$$\hat{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t f^n, t_{n+1/2}) \tag{101}$$

ince this gives us a prediction of $f^{n+\frac{1}{2}}$, we can for $\tilde{f}^{n+\frac{1}{2}}$ try a Backward Euler
ethod to approximate $u^{n+\frac{1}{2}}$:

$$\tilde{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1/2})\,. \tag{102}$$

Vith $\tilde{f}^{n+\frac{1}{2}}$ as a hopefully good approximation to $f^{n+\frac{1}{2}}$, we can for the final
erm $\bar{f}^{n+1}$ use a Crank-Nicolson method to approximate u^{n+1}:

$$\bar{f}^{n+1} = f(u^n + \Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1})\,. \tag{103}$$

[22] http://en.wikipedia.org/wiki/Simpson's_rule

We have now used the Forward and Backward Euler methods as wel
Crank-Nicolson method in the context of Simpson's rule. The hope is t
combination of these methods yields an overall time-stepping scheme fro
t_n+1 that is much more accurate than the $\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta t^2)$ of the in
steps. This is indeed true: the overall accuracy is $\mathcal{O}(\Delta t^4)$!

To summarize, the 4th-order Runge-Kutta method becomes

$$u^{n+1} = u^n + \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right),$$

where the quantities on the right-hand side are computed from (101
Note that the scheme is fully explicit so there is never any need to solv
or nonlinear algebraic equations. However, the stability is conditio
depends on f. There is a whole range of *implicit* Runge-Kutta methods
unconditionally stable, but require solution of algebraic equations involv
each time step.

The simplest way to explore more sophisticated methods for OD
apply one of the many high-quality software packages that exist, as t
section explains.

6.9 The Odespy software

A wide range of the methods and software exist for solving (89). Many of
are accessible through a unified Python interface offered by the Odespy[23]
Odespy features simple Python implementations of the most fundamental
as well as Python interfaces to several famous packages for solving
ODEPACK[24], Vode[25], rkc.f[26], rkf45.f[27], Radau5[28], as well as the ODE
in SciPy[29], SymPy[30], and odelab[31].

The usage of Odespy follows this setup for the ODE $u' = -au$, u
$t \in (0,T]$, here solved by the famous 4th-order Runge-Kutta metho
$\Delta t = 1$ and $N_t = 6$ steps:

```
def f(u, t):
    return -a*u

import odespy
import numpy as np

I = 1; a = 0.5; Nt = 6; dt = 1
solver = odespy.RK4(f)
solver.set_initial_condition(I)
t_mesh = np.linspace(0, Nt*dt, Nt+1)
u, t = solver.solve(t_mesh)
```

[23] https://github.com/hplgit/odespy
[24] https://computation.llnl.gov/casc/odepack/odepack_home.html
[25] https://computation.llnl.gov/casc/odepack/odepack_home.html
[26] http://www.netlib.org/ode/rkc.f
[27] http://www.netlib.org/ode/rkf45.f
[28] http://www.unige.ch/ hairer/software.html
[29] http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.od
[30] http://docs.sympy.org/dev/modules/mpmath/calculus/odes.html
[31] http://olivierverdier.github.com/odelab/

The previously listed methods for ODEs are all accessible in Odespy:

- the θ-rule: `ThetaRule`
- special cases of the θ-rule: `ForwardEuler`, `BackwardEuler`, `CrankNicolson`
- the 2nd- and 4th-order Runge-Kutta methods: `RK2` and `RK4`
- The BDF methods and the Adam-Bashforth methods: `Vode`, `Lsode`, `Lsoda`, `lsoda_scipy`
- The Leapfrog scheme: `Leapfrog` and `LeapfrogFiltered`

.10 Example: Runge-Kutta methods

ince all solvers have the same interface in Odespy, modulo different set of arameters to the solvers' constructors, one can easily make a list of solver bjects and run a loop for comparing (a lot of) solvers. The code below, found n complete form in `decay_odespy.py`[32], compares the famous Runge-Kutta ethods of orders 2, 3, and 4 with the exact solution of the decay equation $' = -au$. Since we have quite long time steps, we have included the only elevant θ-rule for large time steps, the Backward Euler scheme ($\theta = 1$), as well. igure 15 shows the results.

```
import numpy as np
import scitools.std as plt
import sys

def f(u, t):
    return -a*u

I = 1; a = 2; T = 6
dt = float(sys.argv[1]) if len(sys.argv) >= 2 else 0.75
Nt = int(round(T/dt))
t = np.linspace(0, Nt*dt, Nt+1)

solvers = [odespy.RK2(f),
           odespy.RK3(f),
           odespy.RK4(f),
           odespy.BackwardEuler(f, nonlinear_solver='Newton')]

legends = []
for solver in solvers:
    solver.set_initial_condition(I)
    u, t = solver.solve(t)

    plt.plot(t, u)
    plt.hold('on')
    legends.append(solver.__class__.__name__)

# Compare with exact solution plotted on a very fine mesh
t_fine = np.linspace(0, T, 10001)
```

[32] http://tinyurl.com/jvzzcfn/decay/decay_odespy.py


```
u_e = I*np.exp(-a*t_fine)
plt.plot(t_fine, u_e, '-') # avoid markers by specifying line typ
legends.append('exact')

plt.legend(legends)
plt.title('Time step: %g' % dt)
plt.show()
```

Visualization tip.

We use SciTools for plotting here, but importing `matplotlib.pypl` `plt` instead also works. However, plain use of Matplotlib as done results in curves with different colors, which may be hard to distin on black-and-white paper. Using SciTools, curves are automatically colors *and* markers, thus making curves easy to distinguish on screen colors and on black-and-white paper. The automatic adding of mark normally a bad idea for a very fine mesh since all the markers get clut but SciTools limits the number of markers in such cases. For the solution we use a very fine mesh, but in the code above we specify th type as a solid line (`-`), which means no markers and just a color automatically determined by the backend used for plotting (Matpl by default, but SciTools gives the opportunity to use other backen produce the plot, e.g., Gnuplot or Grace).

Also note the that the legends are based on the class names o solvers, and in Python the name of a the class type (as a string) object `obj` is obtained by `obj.__class__.__name__`.

The runs in Figure 15 and other experiments reveal that the 2n Runge-Kutta method (`RK2`) is unstable for $\Delta t > 1$ and decays slower t Backward Euler scheme for large and moderate Δt (see Exercise 7 for an a However, for fine $\Delta t = 0.25$ the 2nd-order Runge-Kutta method app the exact solution faster than the Backward Euler scheme. That is, th scheme does a better job for larger Δt, while the higher order scheme is for smaller Δt. This is a typical trend also for most schemes for ordin partial differential equations.

The 3rd-order Runge-Kutta method (`RK3`) has also artifacts in oscillatory behavior for the larger Δt values, much like that of the Nicolson scheme. For finer Δt, the 3rd-order Runge-Kutta method co quickly to the exact solution.

The 4th-order Runge-Kutta method (`RK4`) is slightly inferior to the B Euler scheme on the coarsest mesh, but is then clearly superior to all t schemes. It is definitely the method of choice for all the tested schemes

Remark about using the θ-rule in Odespy. The Odespy package that the ODE is written as $u' = f(u, t)$ with an f that is possibly nonlin

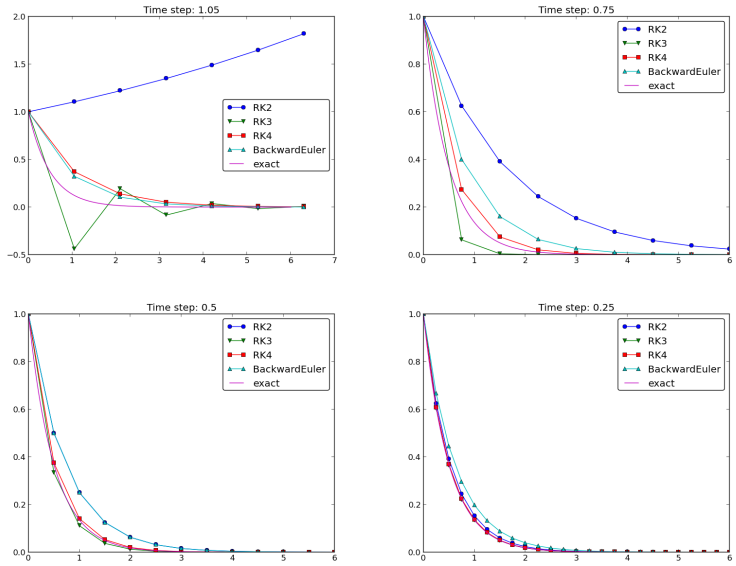


Figure 15: Behavior of different schemes for the decay equation.

he θ-rule for $u' = f(u,t)$ leads to

$$u^{n+1} = u^n + \Delta t \left(\theta f(u^{n+1}, t_{n+1}) + (1-\theta) f(u^n, t_n)\right),$$

hich is a *nonlinear equation* in u^{n+1}. Odespy's implementation of the θ-rule `hetaRule`) and the specialized Backward Euler (`BackwardEuler`) and Crank-icolson (`CrankNicolson`) schemes must invoke iterative methods for solving ıe nonlinear equation in u^{n+1}. This is done even when f is linear in u, as ı the model problem $u' = -au$, where we can easily solve for u^{n+1} by hand. herefore, we need to specify use of Newton's method to the equations. (Odespy llows other methods than Newton's to be used, for instance Picard iteration, ut that method is not suitable. The reason is that it applies the Forward Euler cheme to generate a start value for the iterations. Forward Euler may give very rong solutions for large Δt values. Newton's method, on the other hand, is ısensitive to the start value in *linear problems*.)

.11 Example: Adaptive Runge-Kutta methods

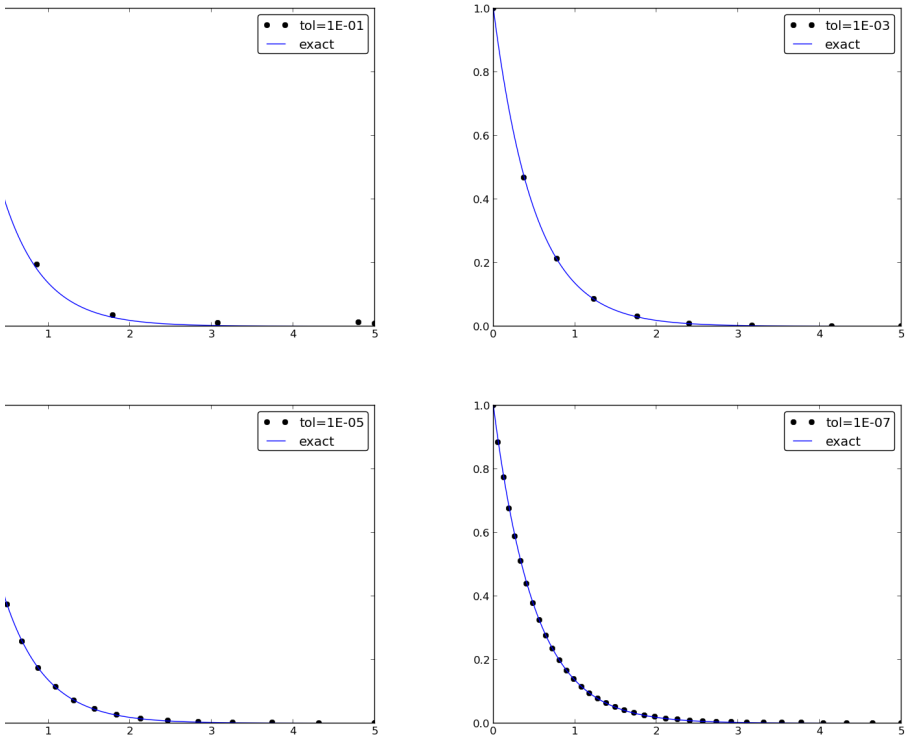
odespy offers solution methods that can adapt the size of Δt with time to match desired accuracy in the solution. Intuitively, small time steps will be chosen in reas where the solution is changing rapidly, while larger time steps can be used here the solution is slowly varying. Some kind of *error estimator* is used to djust the next time step at each time level.

A very popular adaptive method for solving ODEs is the Dormanc Runge-Kutta method of order 4 and 5. The 5th-order method is us reference solution and the difference between the 4th- and 5th-order me used as an indicator of the error in the numerical solution. The Dormanc method is the default choice in MATLAB's widely used `ode45` routine.

We can easily set up Odespy to use the Dormand-Prince method how it selects the optimal time steps. To this end, we request only one ti from $t = 0$ to $t = T$ and ask the method to compute the necessary non- time mesh to meet a certain error tolerance. The code goes like

```
import odespy
import numpy as np
import decay_mod
import sys
#import matplotlib.pyplot as plt
import scitools.std as plt

def f(u, t):
    return -a*u

def exact_solution(t):
    return I*np.exp(-a*t)

I = 1; a = 2; T = 5
tol = float(sys.argv[1])
solver = odespy.DormandPrince(f, atol=tol, rtol=0.1*tol)

Nt = 1  # just one step - let the scheme find its intermediate p
t_mesh = np.linspace(0, T, Nt+1)
t_fine = np.linspace(0, T, 10001)

solver.set_initial_condition(I)
u, t = solver.solve(t_mesh)

# u and t will only consist of [I, u^Nt] and [0,T]
# solver.u_all and solver.t_all contains all computed points
plt.plot(solver.t_all, solver.u_all, 'ko')
plt.hold('on')
plt.plot(t_fine, exact_solution(t_fine), 'b-')
plt.legend(['tol=%.0E' % tol, 'exact'])
plt.savefig('tmp_odespy_adaptive.png')
plt.show()
```

Running four cases with tolerances 10^{-1}, 10^{-3}, 10^{-5}, and 10^{-7}, g results in Figure 16. Intuitively, one would expect denser points in the be of the decay and larger time steps when the solution flattens out.



igure 16: Choice of adaptive time mesh by the Dormand-Prince method for ifferent tolerances.

Exercises

xercise 3: Experiment with precision in tests and the size f u

is claimed in Section 5.5 that most numerical methods will reproduce a linear exct solution to machine precision. Test this assertion using the nose test function `est_linear_solution` in the `decay_vc.py`[33] program. Vary the parameter `c` om very small, via `c=1` to many larger values, and print out the maximum differnce between the numerical solution and the exact solution. What is the relevant alue of the `places` (or `delta`) argument to `nose.tools.assert_almost_equal` n each case? Filename: `test_precision.py`.

[33] `http://tinyurl.com/jvzzcfn/decay/decay_vc.py`

Exercise 4: Implement the 2-step backward scheme

Implement the 2-step backward method (92) for the model $u'(t) = -a($ $b(t)$, $u(0) = I$. Allow the first step to be computed by either the Bа Euler scheme or the Crank-Nicolson scheme. Verify the implementa choosing $a(t)$ and $b(t)$ such that the exact solution is linear in t (see Sect Show mathematically that a linear solution is indeed a solution of the equations.

Compute convergence rates (see Section **??**) in a test case $a = co$ $b = 0$, where we easily have an exact solution, and determine if the c a first-order scheme (Backward Euler) for the first step has any impact overall accuracy of this scheme. The expected error goes like $\mathcal{O}(\Delta t^2)$. F `decay_backward2step.py`.

Exercise 5: Implement the 2nd-order Adams-Bashfort

Implement the 2nd-order Adams-Bashforth method (99) for the decay p $u' = -a(t)u + b(t)$, $u(0) = I$, $t \in (0, T]$. Use the Forward Euler method first step such that the overall scheme is explicit. Verify the impleme using an exact solution that is linear in time. Analyze the scheme by se for solutions $u^n = A^n$ when $a = \text{const}$ and $b = 0$. Compare this secon secheme to the Crank-Nicolson scheme. Filename: `decay_AdamsBashfor`

Exercise 6: Implement the 3rd-order Adams-Bashfort

Implement the 3rd-order Adams-Bashforth method (100) for the decay p $u' = -a(t)u+b(t)$, $u(0) = I$, $t \in (0, T]$. Since the scheme is explicit, allow started by two steps with the Forward Euler method. Investigate experi the case where $b = 0$ and a is a constant: Can we have oscillatory solut large Δt? Filename: `decay_AdamsBashforth3.py`.

Exercise 7: Analyze explicit 2nd-order methods

Show that the schemes (97) and (98) are identical in the case $f(u, t) = -$ $a > 0$ is a constant. Assume that the numerical solution reads $u^n = A^n$ unknown amplification factor A to be determined. Find A and derive s criteria. Can the scheme produce oscillatory solutions of $u' = -au$? I numerical and exact amplification factor. Filename: `decay_RK2_Taylo`

Problem 8: Implement and investigate the Leapfrog sc

A Leapfrog scheme for the ODE $u'(t) = -a(t)u(t) + b(t)$ is defined by

$$[D_{2t}u = -au + b]^n .$$

A separate method is needed to compute u^1. The Forward Euler sche possible candidate.

) Implement the Leapfrog scheme for the model equation. Plot the solution
ı the case $a = 1$, $b = 0$, $I = 1$, $\Delta t = 0.01$, $t \in [0, 4]$. Compare with the exact
olution $u_{\mathrm{e}}(t) = e^{-t}$.

) Show mathematically that a linear solution in t fulfills the Forward Euler
:heme for the first step and the Leapfrog scheme for the subsequent steps. Use
ıis linear solution to verify the implementation, and automate the verification
ırough a nose test.

Hint. It can be wise to automate the calculations such that it is easy to redo
ıe calculations for other types of solutions. Here is a possible `sympy` function
ıat takes a symbolic expression `u` (implemented as a Python function of `t`), fits
ıe `b` term, and checks if `u` fulfills the discrete equations:

```
ımport sympy as sp

lef analyze(u):
    t, dt, a = sp.symbols('t dt a')

    print 'Analyzing u_e(t)=%s' % u(t)
    print 'u(0)=%s' % u(t).subs(t, 0)

    # Fit source term to the given u(t)
    b = sp.diff(u(t), t) + a*u(t)
    b = sp.simplify(b)
    print 'Source term b:', b

    # Residual in discrete equations; Forward Euler step
    R_step1 = (u(t+dt) - u(t))/dt + a*u(t) - b
    R_step1 = sp.simplify(R_step1)
    print 'Residual Forward Euler step:', R_step1

    # Residual in discrete equations; Leapfrog steps
    R = (u(t+dt) - u(t-dt))/(2*dt) + a*u(t) - b
    R = sp.simplify(R)
    print 'Residual Leapfrog steps:', R

lef u_e(t):
    return c*t + I

ınalyze(u_e)
# or short form: analyze(lambda t: c*t + I)
```

) Show that a second-order polynomial in t cannot be a solution of the discrete
quations. However, if a Crank-Nicolson scheme is used for the first step, a
econd-order polynomial solves the equations exactly.

) Create a manufactured solution $u(t) = \sin(t)$ for the ODE $u' = -au + b$.
'ompute the convergence rate of the Leapfrog scheme using this manufactured
olution. The expected convergence rate of the Leapfrog scheme is $\mathcal{O}(\Delta t^2)$. Does
ıe use of a 1st-order method for the first step impact the convergence rate?

e) Set up a set of experiments to demonstrate that the Leapfrog scheme
associated with numerical artifacts (instabilities). Document the main
from this investigation.

f) Analyze and explain the instabilities of the Leapfrog scheme (94):

1. Choose $a = \text{const}$ and $b = 0$. Assume that an exact solution of the
equations has the form $u^n = A^n$, where A is an amplification fa
be determined. Derive an equation for A by inserting $u^n = A^n$
Leapfrog scheme.
2. Compute A either by hand and/or with the aid of `sympy`. The pol
for A has two roots, A_1 and A_2. Let u^n be a linear combinatio
$C_1 A_1^n + C_2 A_2^n$.
3. Show that one of the roots is the explanation of the instability.
4. Compare A with the exact expression, using a Taylor series approx
5. How can C_1 and C_2 be determined?

g) Since the original Leapfrog scheme is unconditionally unstable as tim
it demands some stabilization. This can be done by filtering, where we f
u^{n+1} from the original Leapfrog scheme and then replace u^n by $u^n + \gamma$
$2u^n + u^{n+1}$), where γ can be taken as 0.6. Implement the filtered L
scheme and check that it can handle tests where the original Leapfrog sc
unstable.
Filenames: `decay_leapfrog.py`, `decay_leapfrog.pdf`.

Problem 9: Make a unified implementation of many sc

Consider the linear ODE problem $u'(t) = -a(t)u(t) + b(t)$, $u(0) = I$.
schemes for this problem can be written in the general form

$$u^{n+1} = \sum_{j=0}^{m} c_j u^{n-j},$$

for some choice of $c_0, \ldots, c_m$. Find expressions for the c_j coefficients in
the θ-rule, the three-level backward scheme, the Leapfrog scheme, the 2n
Runge-Kutta method, and the 3rd-order Adams-Bashforth scheme.

Make a class `ExpDecay` that implements the general updating formu
The formula cannot be applied for $n < m$, and for those n values, other
must be used. Assume for simplicity that we just repeat Crank-Nicols
until (105) can be used. Use a subclass to specify the list $c_0, \ldots, c$
particular method, and implement subclasses for all the mentioned s
Verify the implementation by testing with a linear solution, which sh
exactly reproduced by all methods. Filename: `decay_schemes_oo.py`.

8 Applications of exponential decay models

his section presents many mathematical models that all end up with ODEs of ıe type $u' = -au + b$. The applications are taken from biology, finance, and hysics, and cover population growth or decay, compound interest and inflation, ıdioactive decay, cooling of objects, compaction of geological media, pressure ıriations in the atmosphere, and air resistance on falling or rising bodies.

.1 Scaling

eal applications of a model $u' = -au + b$ will often involve a lot of parameters ı the expressions for a and b. It can be quite a challenge to find relevant values f all parameters. In simple problems, however, it turns out that it is not always ecessary to estimate all parameters because we can lump them into one or a w *dimensionless* numbers by using a very attractive technique called scaling. It mply means to stretch the u and t axis is the present problem - and suddenly all arameters in the problem are lumped one parameter if $b \neq 0$ and no parameter hen $b = 0$!

Scaling means that we introduce a new function $\bar{u}(\bar{t})$, with

$$\bar{u} = \frac{u - u_m}{u_c}, \quad \bar{t} = \frac{t}{t_c},$$

here u_m is a characteristic value of u, u_c is a characteristic size of the range of u alues, and t_c is a characteristic size of the range of t_c where u varies significantly. 'hoosing u_m, u_c, and t_c is not always easy and often an art in complicated roblems. We just state one choice first:

$$u_c = I, \quad u_m = b/a, \quad t_c = 1/a\,.$$

ıserting $u = u_m + u_c\bar{u}$ and $t = t_c\bar{t}$ in the problem $u' = -au + b$, assuming a nd b are constants, results after some algebra in the *scaled problem*

$$\frac{d\bar{u}}{d\bar{t}} = -\bar{u}, \quad \bar{u}(0) = 1 - \beta,$$

here β is a dimensionless number

$$\beta = \frac{b}{Ia}\,.$$

hat is, only the special combination of $b/(Ia)$ matters, not what the individual alues of b, a, and I are. Moreover, if $b = 0$, the scaled problem is independent f a and I! In practice this means that we can perform one numerical simulation f the scaled problem and recover the solution of any problem for a given a and by stretching the axis in the plot: $u = I\bar{u}$ and $t = \bar{t}/a$. For $b \neq 0$, we simulate ıe scaled problem for a few β values and recover the physical solution u by ·anslating and stretching the u axis and stretching the t axis.

The scaling breaks down if $I = 0$. In that case we may choose $u_m = 0$, $_c = b/a$, and $t_c = 1/b$, resulting in a slightly different scaled problem:

$$\frac{d\bar{u}}{d\bar{t}} = 1 - \bar{u}, \quad \bar{u}(0) = 0\,.$$

As with $b = 0$, the case $I = 0$ has a scaled problem with no physical para

It is common to drop the bars after scaling and write the scaled p as $u' = -u$, $u(0) = 1 - \beta$, or $u' = 1 - u$, $u(0) = 0$. Any implementatio problem $u' = -au + b$, $u(0) = I$, can be reused for the scaled problem by $a = 1$, $b = 0$, and $I = 1 - \beta$ in the code, if $I \neq 0$, or one sets $a = 1$ and $I = 0$ when the physical I is zero. Falling bodies in fluids, as desc Section 8.8, involves $u' = -au + b$ with seven physical parameters. A vanish in the scaled version of the problem if we start the motion from

8.2 Evolution of a population

Let N be the number of individuals in a population occupying some domain. Despite N being an integer in this problem, we shall compute as a real number and view $N(t)$ as a continuous function of time. Th model assumption is that in a time interval Δt the number of newcomer populations (newborns) is proportional to N, with proportionality co The amount of newcomers will increase the population and result in to

$$N(t + \Delta t) = N(t) + \bar{b}N(t)\,.$$

It is obvious that a long time interval Δt will result in more newcom hence a larger $\bar{b}$. Therefore, we introduce $b = \bar{b}/\Delta t$: the number of new per unit time and per individual. We must then multiply b by the lengt time interval considered and by the population size to get the total nu new individuals, $b\Delta tN$.

If the number of removals from the population (deaths) is also prop to N, with proportionality constant $d\Delta t$, the population evolves accord

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t)\,.$$

Dividing by Δt and letting $\Delta t \to 0$, we get the ODE

$$N' = (b - d)N, \quad N(0) = N_0\,.$$

In a population where the death rate (d) is larger than then newborn $a > 0$, and the population experiences exponential decay rather than exp growth.

In some populations there is an immigration of individuals into the domain. With I individuals coming in per time unit, the equation population change becomes

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t) + \Delta tI\,.$$

The corresponding ODE reads

$$N' = (b - d)N + I, \quad N(0) = N_0\,.$$

Some simplification arises if we introduce a fractional measure of the popula-on: $u = N/N_0$ and set $r = b - d$. The ODE problem now becomes

$$u' = ru + f, \quad u(0) = 1, \tag{108}$$

here $f = I/N_0$ measures the net immigration per time unit as the fraction of ıe initial population. Very often, r is approximately constant, but f is usually function of time.

The growth rate r of a population decreases if the environment has limited esources. Suppose the environment can sustain at most $N_{\max}$ individuals. We ıay then assume that the growth rate approaches zero as N approaches $N_{\max}$, e., as u approaches $M = N_{\max}/N_0$. The simplest possible evolution of r is ıen a linear function: $r(t) = r_0(1 - u(t)/M)$, where r_0 is the initial growth rate hen the population is small relative to the maximum size and there is enough esources. Using this $r(t)$ in (108) results in the *logistic model* for the evolution f a population (assuming for the moment that $f = 0$):

$$u' = r_0(1 - u/M)u, \quad u(0) = 1\,. \tag{109}$$

ıitially, u will grow at rate r_0, but the growth will decay as u approaches M, nd then there is no more change in u, causing $u \to M$ as $t \to \infty$. Note that ıe logistic equation $u' = r_0(1 - u/M)u$ is *nonlinear* because of the quadratic erm $-u^2 r_0/M$.

.3 Compound interest and inflation

ay the annual interest rate is r percent and that the bank adds the interest nce a year to your investment. If u^n is the investment in year n, the investment ı year u^{n+1} grows to

$$u^{n+1} = u^n + \frac{r}{100}u^n\,.$$

ı reality, the interest rate is added every day. We therefore introduce a parameter ı for the number of periods per year when the interest is added. If n counts ıe periods, we have the fundamental model for compound interest:

$$u^{n+1} = u^n + \frac{r}{100m}u^n\,. \tag{110}$$

his model is a *difference equation*, but it can be transformed to a continuous ifferential equation through a limit process. The first step is to derive a formula or the growth of the investment over a time t. Starting with an investment u^0,

and assuming that r is constant in time, we get

$$\begin{aligned} u^{n+1} &= \left(1 + \frac{r}{100m}\right) u^n \\ &= \left(1 + \frac{r}{100m}\right)^2 u^{n-1} \\ &\vdots \\ &= \left(1 + \frac{r}{100m}\right)^{n+1} u^0 \end{aligned}$$

Introducing time t, which here is a real-numbered counter for years, that $n = mt$, so we can write

$$u^{mt} = \left(1 + \frac{r}{100m}\right)^{mt} u^0\,.$$

The second step is to assume *continuous compounding*, meaning that the is added continuously. This implies $m \to \infty$, and in the limit one formula

$$u(t) = u_0 e^{rt/100},$$

which is nothing but the solution of the ODE problem

$$u' = \frac{r}{100}u, \quad u(0) = u_0\,.$$

This is then taken as the ODE model for compound interest if $r > 0$. H the reasoning applies equally well to inflation, which is just the case One may also take the r in (112) as the net growth of an investemt, takes both compound interest and inflation into account. Note that applications we must use a time-dependent r in (112).

Introducing $a = \frac{r}{100}$, continuous inflation of an initial fortune I is process exhibiting exponential decay according to

$$u' = -au, \quad u(0) = I\,.$$

8.4 Radioactive Decay

An atomic nucleus of an unstable atom may lose energy by emitting particles and thereby be transformed to a nucleus with a different nu protons and neutrons. This process is known as radioactive decay[34]. A the process is stochastic when viewed for a single atom, because it is im to predict exactly when a particular atom emits a particle. Neverthele a large number of atoms, N, one may view the process as determinis compute the mean behavior of the decay. Below we reason intuitivel an ODE for the mean behavior. Thereafter, we show mathematically detailed stochastic model for single atoms leads the same mean behavic

[34] `http://en.wikipedia.org/wiki/Radioactive_decay`

Deterministic model. Suppose at time t, the number of the original atom type is $N(t)$. A basic model assumption is that the transformation of the atoms of the original type in a small time interval Δt is proportional to N, so that

$$N(t+\Delta t) = N(t) - a\Delta t N(t),$$

where $a > 0$ is a constant. Introducing $u = N(t)/N(0)$, dividing by Δt and letting $\Delta t \to 0$ gives the following ODE:

$$u' = -au, \quad u(0) = 1\,. \tag{113}$$

The parameter a can for a given nucleus be expressed through the *half-life* $t_{1/2}$, which is the time taken for the decay to reduce the initial amount by one half, i.e., $u(t_{1/2}) = 0.5$. With $u(t) = e^{-at}$, we get $t_{1/2} = a^{-1}\ln 2$ or $a = \ln 2/t_{1/2}$.

Stochastic model. We have originally N_0 atoms. Each atom may have decayed or survived at a particular time t. We want to count how many original atoms that are left, i.e., how many atoms that have survived. The survival of a single atom at time t is a random event. Since there are only two outcomes, survival or decay, we have a Bernoulli trial[35]. Let p be the probability of survival (implying that the probability of decay is $1-p$). If each atom survives independently of the others, and the probability of survival is the same for every atom, we have N_0 statistically Bernoulli trials, known as a *binomial experiment* from probability theory. The probability $P(N)$ that N out of the N_0 atoms have survived at time t is then given by the famous *binomial distribution*

$$P(N) = \frac{N_0!}{N!(N_0-N)!}p^N(1-p)^{N_0-N}\,.$$

The mean (or expected) value $\mathrm{E}[P]$ of $P(N)$ is known to be N_0p.

It remains to estimate p. Let the interval $[0,t]$ be divided into m small subintervals of length Δt. We make the assumption that the probability of decay of a single atom in an interval of length Δt is $\tilde{p}$, and that this probability is proportional to Δt: $\tilde{p} = \lambda\Delta t$ (it sounds natural that the probability of decay increases with Δt). The corresponding probability of survival is $1-\lambda\Delta t$. Believing that λ is independent of time, we have, for each interval of length Δt, a Bernoulli trial: the atom either survives or decays in that interval. Now, p should be the probability that the atom survives in all the intervals, i.e., that we have m successful Bernoulli trials in a row and therefore

$$p = (1-\lambda\Delta t)^m\,.$$

The expected number of atoms of the original type at time t is

$$\mathrm{E}[P] = N_0p = N_0(1-\lambda\Delta t)^m, \quad m = t/\Delta t\,. \tag{114}$$

[35] http://en.wikipedia.org/wiki/Bernoulli_trial

To see the relation between the two types of Bernoulli trials and the above, we go to the limit $\Delta t \to t$, $m \to \infty$. One can show that

$$p = \lim_{m\to\infty}(1-\lambda\Delta t)^m = \lim_{m\to\infty}\left(1-\lambda\frac{t}{m}\right)^m = e^{-\lambda t}$$

This is the famous exponential waiting time (or arrival time) distribution a Poisson process in probability theory (obtained here, as often done limit of a binomial experiment). The probability of decay, $1-e^{-\lambda t}$, follows exponential distribution[36]. The limit means that m is very large, hence very small, and $\tilde{p} = \lambda\Delta t$ is very small since the intensity of the event assumed finite. This situation corresponds to a very small probability atom will decay in a very short time interval, which is a reasonable model same model occurs in lots of different applications, e.g., when waiting for or when finding defects along a rope.

Relation between stochastic and deterministic models. With p we get the expected number of original atoms at t as $N_0p = N_0e^{-\lambda t}$ is exactly the solution of the ODE model $N' = -\lambda N$. This gives interpretation of a via λ or vice versa. Our important finding here is that ODE model captures the mean behavior of the underlying stochastic This is, however, not always the common relation between microscopic stochastic models and macroscopic "averaged" models.

Also of interest is to see that a Forward Euler discretization of N' $N(0) = N_0$, gives $N^m = N_0(1-\lambda\Delta t)^m$ at time $t_m = m\Delta t$, which is the expected value of the stochastic experiment with N_0 atoms and m intervals of length Δt, where each atom can decay with probability $\lambda\Delta$ interval.

A fundamental question is how accurate the ODE model is. The underlying stochastic model fluctuates around its expected value. A measure of the fluctuations is the standard deviation of the binomial experiment with N_0 which can be shown to be $\mathrm{Std}[P] = \sqrt{N_0p(1-p)}$. Compared to the size of the expectation, we get the normalized standard deviation

$$\frac{\sqrt{\mathrm{Var}[P]}}{\mathrm{E}[P]} = N_0^{-1/2}\sqrt{p^{-1}-1} = N_0^{-1/2}\sqrt{(1-e^{-\lambda t})^{-1}-1} \approx (N_0\lambda t)^{-}$$

showing that the normalized fluctuations are very small if N_0 is very large, as is usually the case.

8.5 Newton's law of cooling

When a body at some temperature is placed in a cooling environment, experience shows that the temperature falls rapidly in the beginning, and the

[36] http://en.wikipedia.org/wiki/Exponential_distribution

anges in temperature levels off until the body's temperature equals that of
ie surroundings. Newton carried out some experiments on cooling hot iron
nd found that the temperature evolved as a "geometric progression at times in
rithmetic progression", meaning that the temperature decayed exponentially.
ater, this result was formulated as a differential equation: the rate of change of
ie temperature in a body is proportional to the temperature difference between
ie body and its surroundings. This statement is known as *Newton's law of
ooling*, which can be mathematically expressed as

$$\frac{dT}{dt} = -k(T - T_s), \tag{115}$$

here T is the temperature of the body, T_s is the temperature of the surroundings,
is time, and k is a positive constant. Equation (133) is primarily viewed as
n empirical law, valid when heat is efficiently convected away from the surface
f the body by a flowing fluid such as air at constant temperature T_s. The
eat transfer coefficient k reflects the transfer of heat from the body to the
rroundings and must be determined from physical experiments.

We must obviously have an initial condition $T(0) = T_0$ in addition to the
ooling law (133).

.6 Decay of atmospheric pressure with altitude

ertical equilibrium of air in the atmosphere is governed by the equation

$$\frac{dp}{dz} = -\varrho g\,. \tag{116}$$

ere, $p(z)$ is the air pressure, ϱ is the density of air, and $g = 9.807$ m/s^2 is a
andard value of the acceleration of gravity. (Equation (116) follows directly
om the general Navier-Stokes equations for fluid motion, with the assumption
iat the air does not move.)

The pressure is related to density and temperature through the ideal gas law

$$\varrho = \frac{Mp}{R^*T}, \tag{117}$$

here M is the molar mass of the Earth's air (0.029 kg/mol), R^* is the universal
as constant (8.314 Nm/(mol K)), and T is the temperature. All variables p, ϱ,
nd T vary with the height z. Inserting (117) in (116) results in an ODE with a
ariable coefficient:

$$\frac{dp}{dz} = -\frac{Mg}{R^*T(z)}p\,. \tag{118}$$

Multiple atmospheric layers. The atmosphere can be approximately mod-
ed by seven layers. In each layer, (118) is applied with a linear temperature of
ie form

$$T(z) = \bar{T}_i + L_i(z - h_i),$$

where $z = h_i$ denotes the bottom of layer number i, having temperature
L_i is a constant in layer number i. The table below lists h_i (m), $\bar{T}_i$ (K),
(K/m) for the layers $i = 0, \ldots, 6$.

i	h_i	$\bar{T}_i$	L_i
0	0	288	-0.0065
1	11,000	216	0.0
2	20,000	216	0.001
3	32,000	228	0.0028
4	47,000	270	0.0
5	51,000	270	-0.0028
6	71,000	214	-0.002

For implementation it might be convenient to write (118) on the form

$$\frac{dp}{dz} = -\frac{Mg}{R^*(\bar{T}(z) + L(z)(z - h(z)))}p,$$

where $\bar{T}(z)$, $L(z)$, and $h(z)$ are piecewise constant functions with values
the table. The value of the pressure at the sea level $z = 0$, $p_0 = p(0)$, is
Pa.

Simplification: $L = 0$. One commonly used simplification is to assu
the temperature is constant within each layer. This means that $L = 0$.

Simplification: one-layer model. Another commonly used approx
is to work with one layer instead of seven. This one-layer model[37] is b
$T(z) = T_0 - Lz$, with sea level standard temperature $T_0 = 288$ K and temp
lapse rate $L = 0.0065$ K/m.

8.7 Compaction of sediments

Sediments, originally made from materials like sand and mud, get con
through geological time by the weight of new material that is deposited
sea bottom. The porosity ϕ of the sediments tells how much void (fluid
there is between the sand and mud grains. The porosity reduces with
because the weight of the sediments above and causes the void space t
and thereby increase the compaction.

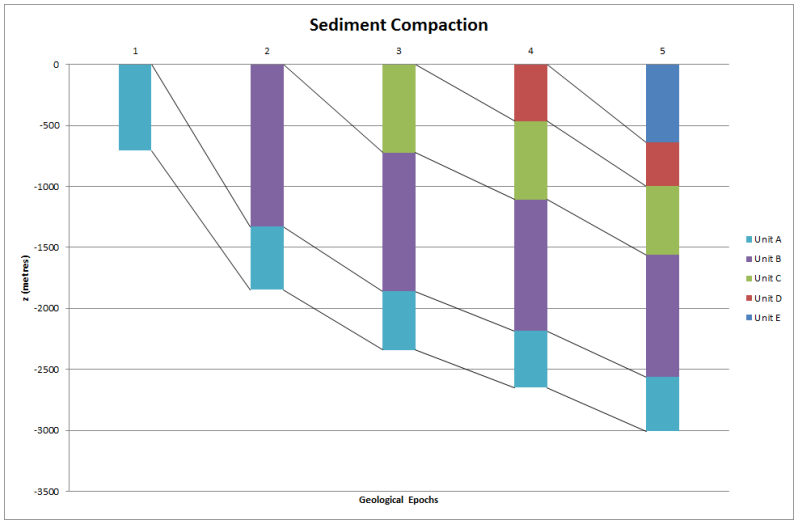
A typical assumption is that the change in ϕ at some depth z is ne
proportional to ϕ. This assumption leads to the differential equation pr

$$\frac{d\phi}{dz} = -c\phi, \quad \phi(0) = \phi_0,$$

[37] `http://en.wikipedia.org/wiki/Density_of_air`

here the z axis points downwards, $z = 0$ is the surface with known porosity, nd $c > 0$ is a constant.

The upper part of the Earth's crust consists of many geological layers stacked n top of each other, as indicated in Figure 17. The model (120) can be applied or each layer. In layer number i, we have the unknown porosity function $\phi_i(z)$ ulfilling $\phi_i'(z) = -c_i z$, since the constant c in the model (120) depends on ne type of sediment in the layer. From the figure we see that new layers of ediments are deposited on top of older ones as time progresses. The compaction, s measured by ϕ, is rapid in the beginning and then decreases (exponentially) ith depth in each layer.



igure 17: Illustration of the compaction of geological layers (with different olors) through time.

When we drill a well at present time through the right-most column of ediments in Figure 17, we can measure the thickness of the sediment in (say) the ottom layer. Let L_1 be this thickness. Assuming that the volume of sediment emains constant through time, we have that the initial volume, $\int_0^{L_{1,0}} \phi_1 dz$, must qual the volume seen today, $\int_{\ell-L_1}^{\ell} \phi_1 dz$, where ℓ is the depth of the bottom of ne sediment in the present day configuration. After having solved for ϕ_1 as a unction of z, we can then find the original thickness $L_{1,0}$ of the sediment from ne equation

$$\int_0^{L_{1,0}} \phi_1 dz = \int_{\ell-L_1}^{\ell} \phi_1 dz\,.$$

n hydrocarbon exploration it is important to know $L_{1,0}$ and the compaction istory of the various layers of sediments.

8.8 Vertical motion of a body in a viscous fluid

A body moving vertically through a fluid (liquid or gas) is subject t different types of forces: the gravity force, the drag force[38], and the bu force.

Overview of forces. The gravity force is $F_g = -mg$, where m is tl of the body and g is the acceleration of gravity. The uplift or buoyan ("Archimedes force") is $F_b = \varrho g V$, where ϱ is the density of the fluid a the volume of the body. Forces and other quantities are taken as positiv upward direction.

The drag force is of two types, depending on the Reynolds number

$$\mathrm{Re} = \frac{\varrho d|v|}{\mu},$$

where d is the diameter of the body in the direction perpendicular to the is the velocity of the body, and μ is the dynamic viscosity of the fluid Re < 1, the drag force is fairly well modeled by the so-called Stokes' drag for a spherical body of diameter d reads

$$F_d^{(S)} = -3\pi d\mu v\,.$$

For large Re, typically Re $> 10^3$, the drag force is quadratic in the velo

$$F_d^{(q)} = -\frac{1}{2} C_D \varrho A|v|v,$$

where C_D is a dimensionless drag coefficient depending on the body's and A is the cross-sectional area as produced by a cut plane, perpendi the motion, through the thickest part of the body. The superscripts ^{q} a $F_d^{(S)}$ and $F_d^{(q)}$ indicate Stokes drag and quadratic drag, respectively.

Equation of motion. All the mentioned forces act in the vertical d Newton's second law of motion applied to the body says that the sum forces must equal the mass of the body times its acceleration a in the direction.

$$ma = F_g + F_d^{(S)} + F_b\,.$$

Here we have chosen to model the fluid resistance by the Stokes drag. I the expressions for the forces yields

$$ma = -mg - 3\pi d\mu v + \varrho g V\,.$$

The unknowns here are v and a, i.e., we have two unknowns but o equation. From kinematics in physics we know that the acceleration is t

[38] http://en.wikipedia.org/wiki/Drag_(physics)

erivative of the velocity: $a = dv/dt$. This is our second equation. We can easily iminate a and get a single differential equation for v:

$$m\frac{dv}{dt} = -mg - 3\pi d\mu v + \varrho g V\,.$$

small rewrite of this equation is handy: We express m as $\varrho_b V$, where ϱ_b is the ensity of the body, and we divide by the mass to get

$$v'(t) = -\frac{3\pi d\mu}{\varrho_b V}v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,. \tag{124}$$

/e may introduce the constants

$$a = \frac{3\pi d\mu}{\varrho_b V}, \quad b = g\left(\frac{\varrho}{\varrho_b} - 1\right), \tag{125}$$

o that the structure of the differential equation becomes obvious:

$$v'(t) = -av(t) + b\,. \tag{126}$$

he corresponding initial condition is $v(0) = v_0$ for some prescribed starting elocity v_0.

This derivation can be repeated with the quadratic drag force $F_d^{(q)}$, leading o the result

$$v'(t) = -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,. \tag{127}$$

efining

$$a = \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}, \tag{128}$$

nd b as above, we can write (127) as

$$v'(t) = -a|v|v + b\,. \tag{129}$$

erminal velocity. An interesting aspect of (126) and (129) is whether v will pproach a final constant value, the so-called *terminal velocity* v_T, as $t \to \infty$. A onstant v means that $v'(t) \to 0$ as $t \to \infty$ and therefore the terminal velocity $_T$ solves

$$0 = -av_T + b$$

nd

$$0 = -a|v_T|v_T + b\,.$$

he former equation implies $v_T = b/a$, while the latter has solutions $v_T = \sqrt{|b|/a}$ for a falling body ($v_T < 0$) and $v_T = \sqrt{b/a}$ for a rising body ($v_T > 0$).

A Crank-Nicolson scheme. Both governing equations, the Stok model (126) and the quadratic drag model (129), can be readily solvec Forward Euler scheme. For higher accuracy one can use the Crank-N method, but a straightforward application this method results a n equation in the new unknown value v^{n+1} when applied to (129):

$$\frac{v^{n+1} - v^n}{\Delta t} = -a\frac{1}{2}(|v^{n+1}|v^{n+1} + |v^n|v^n) + b\,.$$

However, instead of approximating the term $-|v|v$ by an arithmetic ave can use a *geometric mean*:

$$(|v|v)^{n+\frac{1}{2}} \approx |v^n|v^{n+1}\,.$$

The error is of second order in Δt, just as for the arithmetic average centered finite difference approximation in (130). With this approximati the discrete equation

$$\frac{v^{n+1} - v^n}{\Delta t} = -a|v^n|v^{n+1} + b$$

becomes a *linear* equation in v^{n+1}, and we can therefore easily solve fo

$$v^{n+1} = \frac{v_n + \Delta t b^{n+\frac{1}{2}}}{1 + \Delta t a^{n+\frac{1}{2}}|v^n|}\,.$$

Physical data. Suitable values of μ are $1.8\cdot10^{-5}$ Pa s for air and $8.9\cdot1$ for water. Densities can be taken as 1.2 kg/m^3 for air and as $1.0\cdot10^3$ for water. For considerable vertical displacement in the atmosphere on take into account that the density of air varies with the altitude, see Sec One possible density variation arises from the one-layer model in the m section.

Any density variation makes b time dependent and we need $b^{n+\frac{1}{2}}$ i To compute the density that enters $b^{n+\frac{1}{2}}$ we must also compute the position $z(t)$ of the body. Since $v = dz/dt$, we can use a centered di approximation:

$$\frac{z^{n+\frac{1}{2}} - z^{n-\frac{1}{2}}}{\Delta t} = v^n \quad \Rightarrow \quad z^{n+\frac{1}{2}} = z^{n-\frac{1}{2}} + \Delta t\, v^n\,.$$

This $z^{n+\frac{1}{2}}$ is used in the expression for b to compute $\varrho(z^{n+\frac{1}{2}})$ and then

The drag coefficient[39] C_D depends heavily on the shape of the bod values are: 0.45 for a sphere, 0.42 for a semi-sphere, 1.05 for a cube, 0. long cylinder (when the center axis is in the vertical direction), 0.75 for a 1.0-1.3 for a man in upright position, 1.3 for a flat plate perpendicula flow, and 0.04 for a streamlined, droplet-like body.

[39] `http://en.wikipedia.org/wiki/Drag_coefficient`

erification. To verify the program, one may assume a heavy body in air ıch that the F_b force can be neglected, and further assume a small velocity ıch that the air resistance F_d can also be neglected. This can be obtained by ɛtting μ and ϱ to zero. The motion then leads to the velocity $v(t) = v_0 - gt$, hich is linear in t and therefore should be reproduced to machine precision ay tolerance 10^{-15}) by any implementation based on the Crank-Nicolson or orward Euler schemes.

Another verification, but not as powerful as the one above, can be based on ɔmputing the terminal velocity and comparing with the exact expressions. The dvantage of this verification is that we can also the test situation $\varrho \neq 0$.

As always, the method of manufactured solutions can be applied to test the nplementation of all terms in the governing equation, but the solution then has o physical relevance in general.

caling. Applying scaling, as described in Section 8.1, will for the linear case ɛduce the need to estimate values for seven parameters down to choosing one ılue of a single dimensionless parameter

$$\beta = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu I},$$

rovided $I \neq 0$. If the motion starts from rest, $I = 0$, the scaled problem $' = 1 - \bar{u}$, $\bar{u}(0) = 0$, has no need for estimating physical parameters. This means ıat there is a single universal solution to the problem of a falling body starting om rest: $\bar{u}(t) = 1 - e^{-\bar{t}}$. All real physical cases correspond to stretching the $\bar{t}$ xis and the $\bar{u}$ axis in this dimensionless solution. More precisely, the physical elocity $u(t)$ is related to the dimensionless velocity $\bar{u}(\bar{t})$ through

$$u = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu} \bar{u}(t/(g(\varrho/\varrho_b - 1))) .$$

.9 Decay ODEs from solving a PDE by Fourier expansions

uppose we have a partial differential equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f(x,t),$$

ith boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = (x)$. One may express the solution as

$$u(x,t) = \sum_{k=1}^{m} A_k(t) e^{ikx\pi/L},$$

ɔr appropriate unknown functions A_k, $k = 1, \ldots, m$. We use the complex ɛponential $e^{ikx\pi/L}$ for easy algebra, but the physical u is taken as the real

part of any complex expression. Note that the expansion in terms of e^i compatible with the boundary conditions: all functions $e^{ikx\pi/L}$ vanish fo and $x = L$. Suppose we can express $I(x)$ as

$$I(x) = \sum_{k=1}^{m} I_k e^{ikx\pi/L} .$$

Such an expansion can be computed by well-known Fourier expansion tec but the details are not important here. Also, suppose we can express tl $f(x,t)$ as

$$f(x,t) = \sum_{k=1}^{m} b_k(t) e^{ikx\pi/L} .$$

Inserting the expansions for u and f in the differential equations dema all terms corresponding to a given k must be equal. The calculations re the follow system of ODEs:

$$A_k'(t) = -\alpha \frac{k^2\pi^2}{L^2} + b_k(t), \quad k = 1, \ldots, m .$$

From the initial condition

$$u(x,0) = \sum_k A_k(0) e^{ikx\pi/L} = I(x) = \sum_k I_k e^{(ikx\pi/L)},$$

it follows that $A_k(0) = I_k$, $k = 1, \ldots, m$. We then have m equations of t $A_k' = -aA_k + b$, $A_k(0) = I_k$, for appropriate definitions of a and b. The problems independent each other such that we can solve one problem at The outline technique is a quite common approach for solving partial dif equations.

Remark. Since a_k depends on k and the stability of the Forward Euler demands $a_k \Delta t \leq 1$, we get that $\Delta t \leq \alpha^{-1} L^2 \pi^{-2} k^{-2}$. Usually, quite values are needed to accurately represent the given functions I and f and needs to be very small for these large values of k. Therefore, the Crank-N and Backward Euler schemes, which allow larger Δt without any growt solutions, are more popular choices when creating time-stepping algorit partial differential equations of the type considered in this example.

9 Exercises

Exercise 10: Derive schemes for Newton's law of coo

Show in detail how we can apply the ideas of the Forward Euler, Ba Euler, Crank-Nicolson, and θ-rule discretizations to derive explicit compu formulas for new temperature values in Newton's law of cooling (see Sect

$$\frac{dT}{dt} = -k(T - T_s), \quad T(0) = T_0\,. \tag{133}$$

ere, T is the temperature of the body, T_s is the temperature of the surroundings, is time, k is the heat transfer coefficient, and T_0 is the initial temperature of ıe body. Filename: `schemes_cooling.pdf`.

xercise 11: Implement schemes for Newton's law of cool-ıg

ormulate a θ-rule for the three schemes in Exercise 10 such that you can get the ıree schemes from a single formula by varying the θ parameter. Implement the ıethod in a function `cooling(T0, k, T_s, t_end, dt, theta=0.5)`, where ɔ is the initial temperature, `k` is the heat transfer coefficient, `T_s` is the tem-erature of the surroundings, `t_end` is the end time of the simulation, `dt` is the me step, and `theta` corresponds to θ. The `cooling` function should return the ɜmperature as an array `T` of values at the mesh points and the time mesh `t`. onstruct verification examples to check that the implementation works.

lint. For verification, try to find an exact solution of the discrete equations. trick is to introduce $u = T - T_s$, observe that $u^n = (T_0 - T_s)A^n$ for some mplification factor A, and then express this formula in terms of T^n.
ilename: `cooling.py`.

xercise 12: Find time of murder from body temperature

detective measures the temperature of a dead body to be 26.7 C at 2 pm. One our later the temperature is 25.8 C. The question is when death occurred.

Assume that Newton's law of cooling (133) is an appropriate mathematical ıodel for the evolution of the temperature in the body. First, determine k ı (133) by formulating a Forward Euler approximation with one time steep om time 2 am to time 3 am, where knowing the two temperatures allows for nding k. Assume the temperature in the air to be 20 C. Thereafter, simulate ıe temperature evolution from the time of murder, taken as $t = 0$, when ' = 37 C, until the temperature reaches 25.8 C. The corresponding time allows ɔr answering when death occurred. Filename: `detective.py`.

xercise 13: Simulate an oscillating cooling process

he surrounding temperature T_s in Newton's law of cooling (133) may vary in me. Assume that the variations are periodic with period P and amplitude a round a constant mean temperature T_m:

$$T_s(t) = T_m + a\sin\left(\frac{2\pi}{P}t\right)\,. \tag{134}$$

Simulate a process with the following data: $k = 20$ min^{-1}, $T(0) = 5$ C, T C, $a = 2.5$ C, and $P = 1$ h. Also experiment with $P = 10$ min and I Plot T and T_s in the same plot. Filename: `osc_cooling.py`.

Exercise 14: Radioactive decay of Carbon-14

The Carbon-14[40] isotope, whose radioactive decay is used extensively ir organic material that is tens of thousands of years old, has a half-life (years. Determine the age of an organic material that contains 8.4 perce initial amount of Carbon-14. Use a time unit of 1 year in the computatio uncertainty in the half time of Carbon-14 is ±40 years. What is the corres uncertainty in the estimate of the age?

Hint. Use simulations with $5,730 \pm 40$ y as input and find the corres interval for the result.
Filename: `carbon14.py`.

Exercise 15: Simulate stochastic radioactive decay

The purpose of this exercise is to implement the stochastic model desc Section 8.4 and show that its mean behavior approximates the solutio corresponding ODE model.

The simulation goes on for a time interval $[0, T]$ divided into N_t int length Δt. We start with N_0 atoms. In some time interval, we have N that have survived. Simulate N Bernoulli trials with probability $\lambda\Delta t$ interval by drawing N random numbers, each being 0 (survival) or 1 where the probability of getting 1 is $\lambda\Delta t$. We are interested in the of decays, d, and the number of survived atoms in the next interval $N - d$. The Bernoulli trials are simulated by drawing N uniformly dist real numbers on $[0, 1]$ and saying that 1 corresponds to a value less tha

```
# Given lambda_, dt, N
import numpy as np
uniform = np.random.uniform(N)
Bernoulli_trials = np.asarray(uniform < lambda_*dt, dtype=np.int)
d = Bernoulli_trials.size
```

Observe that `uniform < lambda_*dt` is a boolean array whose true a values become 1 and 0, respectively, when converted to an integer arra

Repeat the simulation over $[0, T]$ a large number of times, comp average value of N in each interval, and compare with the solutio corresponding ODE model. Filename: `stochastic_decay.py`.

[40] `http://en.wikipedia.org/wiki/Carbon-14`

Exercise 16: Radioactive decay of two substances

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a half-life $A_{1/2}$, while substance B decay to form type A nuclei with a half-life $B_{1/2}$. Letting u_A and u_B be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$\frac{1}{\ln 2}u'_A = u_B/B_{1/2} - u_A/A_{1/2}, \tag{135}$$

$$\frac{1}{\ln 2}u'_B = u_A/A_{1/2} - u_B/B_{1/2}, \tag{136}$$

with $u_A(0) = u_B(0) = 1$.

Make a simulation program that solves for $u_A(t)$ and $u_B(t)$. Verify the implementation by computing analytically the limiting values of u_A and u_B as $t \to \infty$ (assume $u'_A, u'_B \to 0$) and comparing these with those obtained numerically.

Run the program for the case of $A_{1/2} = 10$ minutes and $B_{1/2} = 50$ minutes. Use a time unit of 1 minute. Plot u_A and u_B versus time in the same plot. Filename: `radioactive_decay_2subst.py`.

Exercise 17: Simulate the pressure drop in the atmosphere

We consider the models for atmospheric pressure in Section 8.6. Make a program with three functions,

- one computing the pressure $p(z)$ using a seven-layer model and varying L,
- one computing $p(z)$ using a seven-layer model, but with constant temperature in each layer, and
- one computing $p(z)$ based on the one-layer model.

How can these implementations be verified? Should ease of verification impact how you code the functions? Compare the three models in a plot. Filename: `atmospheric_pressure.py`.

Exercise 18: Make a program for vertical motion in a fluid

Implement the Stokes' drag model (124) and the quadratic drag model (127) from Section 8.8, using the Crank-Nicolson scheme and a geometric mean for $|v|v$ as explained, and assume constant fluid density. At each time level, compute the Reynolds number Re and choose the Stokes' drag model if $\mathrm{Re} < 1$ and the quadratic drag model otherwise.

The computation of the numerical solution should take place either in a stand-alone function (as in Section 2.1) or in a solver class that looks up a problem class for physical data (as in Section **??**). Create a module (see Section **??**) and equip it with nose tests (see Section **??**) for automatically verifying the code.

Verification tests can be based on

- the terminal velocity (see Section 8.8),
- the exact solution when the drag force is neglected (see Section 8
- the method of manufactured solutions (see Section 5.5) combin computing convergence rates (see Section **??**).

Use, e.g., a quadratic polynomial for the velocity in the method of manu solutions. The expected error is $\mathcal{O}(\Delta t^2)$ from the centered finite di approximation and the geometric mean approximation for $|v|v$.

A solution that is linear in t will also be an exact solution of the equations in many problems. Show that this is true for linear drag (by a source term that depends on t), but not for quadratic drag becaus geometric mean approximation. Use the method of manufactured solu add a source term *in the discrete equations for quadratic drag* such that function of t is a solution. Add a nose test for checking that the linear f is reproduced to machine precision in the case of both linear and quadra

Apply the software to a case where a ball rises in water. The buoyan is here the driving force, but the drag will be significant and balance tl forces after a short time. A soccer ball has radius 11 cm and mass 0.43 k the motion from rest, set the density of water, ϱ, to 1000 kg/m^3, set the viscosity, μ, to 10^{-3} Pa s, and use a drag coefficient for a sphere: 0.45. velocity of the rising ball. Filename: `vertical_motion.py`.

Project 19: Simulate parachuting

The aim of this project is to develop a general solver for the vertical mot body with quadratic air drag, verify the solver, apply the solver to a sky free fall, and finally apply the solver to a complete parachute jump.

All the pieces of software implemented in this project should be rea Python functions and/or classes and collected in one module.

a) Set up the differential equation problem that governs the velocity motion. The parachute jumper is subject to the gravity force and a qu drag force. Assume constant density. Add an extra source term be program verification. Identify the input data to the problem.

b) Make a Python module for computing the velocity of the motion. Al the module with functionality for plotting the velocity.

Hint 1. Use the Crank-Nicolson scheme with a geometric mean of $|v|v$ to linearize the equation of motion with quadratic drag.

Hint 2. You can either use functions or classes for implementation. choose functions, make a function `solver` that takes all the input dat problem as arguments and that returns the velocity (as a mesh functi the time mesh. In case of a class-based implementation, introduce a

lass with the physical data and a solver class with the numerical data and a `olve` method that stores the velocity and the mesh in the class.

Allow for a time-dependent area and drag coefficient in the formula for the rag force.

) Show that a linear function of t does not fulfill the discrete equations because f the geometric mean approximation used for the quadratic drag term. Fit source term, as in the method of manufactured solutions, such that a linear nction of t is a solution of the discrete equations. Make a nose test to check nat this solution is reproduced to machine precision.

) The expected error in this problem goes like Δt^2 because we use a centered nite difference approximation with error $\mathcal{O}(\Delta t^2)$ and a geometric mean approximation with error $\mathcal{O}(\Delta t^2)$. Use the method of manufactured solutions ombined with computing convergence rate to verify the code. Make a nose test or checking that the convergence rate is correct.

) Compute the drag force, the gravity force, and the buoyancy force as a nction of time. Create a plot with these three forces.

lint. You can either make a function `forces(v, t, plot=None)` that returns ne forces (as mesh functions) and `t` and shows a plot on the screen and also aves the plot to a file with name `plot` if `plot` is not `None`, or you can extend ne solver class with computation of forces and include plotting of forces in the isualization class.

) Compute the velocity of a skydiver in free fall before the parachute opens.

lint. Meade and Struthers [5] provide some data relevant to skydiving[41]. The lass of the human body and equipment can be set to 100 kg. A skydiver in pread-eagle formation has a cross-section of 0.5 m^2 in the horizontal plane. The ensity of air decreases varies altitude, but can be taken as constant, 1 kg/m^3, or altitudes relevant to skydiving (0-4000 m). The drag coefficient for a man in pright position can be set to 1.2. Start with a zero velocity. A free fall typically as a terminating velocity of 45 m/s. (This value can be used to tune other arameters.)

) The next task is to simulate a parachute jumper during free fall and after ne parachute opens. At time t_p, the parachute opens and the drag coefficient nd the cross-sectional area change dramatically. Use the program to simulate a ump from $z = 3000$ m to the ground $z = 0$. What is the maximum acceleration, neasured in units of g, experienced by the jumper?

[41] `http://en.wikipedia.org/wiki/Parachuting`

Hint. Following Meade and Struthers [5], one can set the cross-secti perpendicular to the motion to 44 m^2 when the parachute is open. that it takes 8 s to increase the area linearly from the original to the fin The drag coefficient for an open parachute can be taken as 1.8, but tune the known value of the typical terminating velocity reached before land m/s. One can take the drag coefficient as a piecewise constant functi an abrupt change at t_p. The parachute is typically released after $t_p = 6$ larger values of t_p can be used to make plots more illustrative.
Filename: `skydiving.py`.

Exercise 20: Formulate vertical motion in the atmos

Vertical motion of a body in the atmosphere needs to take into account a air density if the range of altitudes is many kilometers. In this case, ϱ var the altitude z. The equation of motion for the body is given in Section us assume quadratic drag force (otherwise the body has to be very, very A differential equation problem for the air density, based on the informa the one-layer atmospheric model in Section 8.6, can be set up as

$$p'(z) = -\frac{Mg}{R^*(T_0 + Lz)}p,$$
$$\varrho = p\frac{M}{R^*T}.$$

To evaluate $p(z)$ we need the altitude z. From the principle that the ve the derivative of the position we have that

$$z'(t) = v(t),$$

where v is the velocity of the body.

Explain in detail how the governing equations can be discretized by ward Euler and the Crank-Nicolson methods. Filename: `falling_in_var`

Exercise 21: Simulate vertical motion in the atmospl

Implement the Forward Euler or the Crank-Nicolson scheme derived cise 20. Demonstrate the effect of air density variation on a falling hum the famous fall of Felix Baumgartner[42]. The drag coefficient can be set

Remark. In the Crank-Nicolson scheme one must solve a 3×3 sy equations at each time level, since p, ϱ, and v are coupled, while each e can be stepped forward at a time with the Forward Euler scheme. Fi `falling_in_variable_density.py`.

[42] `http://en.wikipedia.org/wiki/Felix_Baumgartner`

Exercise 22: Compute $y = |x|$ by solving an ODE

Consider the ODE problem

$$y'(x) = \begin{cases} -1, & x < 0, \\ 1, & x \geq 0 \end{cases} \qquad x \in (-1, 1], \quad y(1-) = 1,$$

which has the solution $y(x) = |x|$. Using a mesh $x_0 = -1$, $x_1 = 0$, and $x_2 = 1$, calculate by hand y_1 and y_2 from the Forward Euler, Backward Euler, Crank-Nicolson, and Leapfrog methods. Use all of the former three methods for computing the y_1 value to be used in the Leapfrog calculation of y_2. Thereafter, visualize how these schemes perform for a uniformly partitioned mesh with $N = 10$ and $N = 11$ points. Filename: `signum.py`.

Exercise 23: Simulate growth of a fortune with random interest rate

The goal of this exercise is to compute the value of a fortune subject to inflation and a random interest rate. Suppose that the inflation is constant at i percent per year and that the annual interest rate, p, changes randomly at each time step, starting at some value p_0 at $t = 0$. The random change is from a value p^n at $t = t_n$ to $p_n + \Delta p$ with probability 0.25 and $p_n - \Delta p$ with probability 0.25. No change occurs with probability 0.5. There is also no change if p^{n+1} exceeds 15 or becomes below 1. Use a time step of one month, $p_0 = i$, initial fortune scaled to 1, and simulate 1000 scenarios of length 20 years. Compute the mean evolution of one unit of money and the corresponding standard deviation. Plot the mean curve along with the mean plus one standard deviation and the mean minus one standard deviation. This will illustrate the uncertainty in the mean curve.

Hint 1. The following code snippet computes p^{n+1}:

```
import random

def new_interest_rate(p_n, dp=0.5):
    r = random.random()  # uniformly distr. random number in [0,1)
    if 0 <= r < 0.25:
        p_np1 = p_n + dp
    elif 0.25 <= r < 0.5:
        p_np1 = p_n - dp
    else:
        p_np1 = p_n
    return (p_np1 if 1 <= p_np1 <= 15 else p_n)
```

Hint 2. If $u_i(t)$ is the value of the fortune in experiment number i, $i = 0, \ldots, N-1$, the mean evolution of the fortune is

$$\bar{u}(t) = \frac{1}{N} \sum_{i=0}^{N-1} u_i(t),$$

and the standard deviation is

$$s(t) = \sqrt{\frac{1}{N-1}\left(-(\bar{u}(t))^2 + \sum_{i=0}^{N-1}(u_i(t))^2\right)}.$$

Suppose $u_i(t)$ is stored in an array `u`. The mean and the standard de of the fortune is most efficiently computed by using two accumulation `sum_u` and `sum_u2`, and performing `sum_u += u` and `sum_u2 += u**2` aft experiment. This technique avoids storing all the $u_i(t)$ time series for co the statistics.
Filename: `random_interest.py`.

Exercise 24: Simulate a population in a changing en ment

We shall study a population modeled by (108) where the environment, repr by r and f, undergoes changes with time.

a) Assume that there is a sudden drop (increase) in the birth (death) time $t = t_r$, because of limited nutrition or food supply:

$$a(t) = \begin{cases} r_0, & t < t_r, \\ r_0 - A, & t \geq t_r, \end{cases}$$

This drop in population growth is compensated by a sudden net immigr time $t_f > t_r$:

$$f(t) = \begin{cases} 0, & t < t_f, \\ f_0, & t \geq t_a, \end{cases}$$

Start with r_0 and make $A > r_0$. Experiment with these and other par to illustrate the interplay of growth and decay in such a problem. Fi `population_drop.py`.

b) Now we assume that the environmental conditions changes periodica time so that we may take

$$r(t) = r_0 + A \sin\left(\frac{2\pi}{P}t\right).$$

That is, the combined birth and death rate oscillates around r_0 with a m change of $\pm A$ repeating over a period of length P in time. Set f = experiment with the other parameters to illustrate typical features of the Filename: `population_osc.py`.

Exercise 25: Simulate logistic growth

Solve the logistic ODE (109) using a Crank-Nicolson scheme where $(u'$ approximated by a *geometric mean*:

$$(u^{n+\frac{1}{2}})^2 \approx u^{n+1}u^n.$$

This trick makes the discrete equation linear in u^{n+1}. Filename: `logistic`

Exercise 26: Rederive the equation for continuous compound interest

The ODE model (112) was derived under the assumption that r was constant. Perform an alternative derivation without this assumption: 1) start with (110); 2) introduce a time step Δt instead of m: $\Delta t = 1/m$ if t is measured in years; 3) divide by Δt and take the limit $\Delta t \to 0$. Simulate a case where the inflation is at a constant level I percent per year and the interest rate oscillates: $r = -I/2 + r_0 \sin(2\pi t)$. Compare solutions for $r_0 = I, 3I/2, 2I$. Filename: `interest_modeling.py`.

References

[1] D. Griffiths, F. David, and D. J. Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. Springer, 2010.

[2] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer, 1993.

[3] G. Hairer and E. Wanner. *Solving Ordinary Differential Equations II*. Springer, 2010.

[4] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Computational Science and Engineering. Springer, third edition, 2012.

[5] D. B. Meade and A. A. Struthers. Differential equations in the new millenium: the parachute problem. *International Journal of Engineering Education*, 15(6):417–424, 1999.

[6] L. Petzold and U. M. Ascher. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, volume 61. SIAM, 1998.

Index